US006571013B1

(12) **United States Patent**
Macey et al.

(10) **Patent No.: US 6,571,013 B1**
(45) **Date of Patent: *May 27, 2003**

(54) **AUTOMATIC METHOD FOR DEVELOPING CUSTOM ICR ENGINES**

(75) Inventors: **Garrett N. Macey**, Rockville, MD (US); **Ivan N. Bella**, Gaithersburg, MD (US)

(73) Assignee: **Lockhead Martin Mission Systems**, Gaithersburg, MD (US)

(*) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/664,221**

(22) Filed: **Jun. 11, 1996**

(51) **Int. Cl.**[7] **G06K 9/00**

(52) **U.S. Cl.** **382/181**; 382/187

(58) **Field of Search** 382/181, 190, 382/192, 201, 209, 159, 226, 160, 161; 235/435

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,054,094	A *	10/1991	Barski	382/192
5,317,652	A *	5/1994	Chatterjee	382/304
5,321,773	A *	6/1994	Kopec et al.	382/159
5,526,444	A *	6/1996	Kopec et al.	382/233
5,594,809	A *	1/1997	Kopec	382/161

OTHER PUBLICATIONS

Mori et al., "Historical Review of OCR Research and Development", Proceedings of the IEEE, vol. 80, No. 7, 7/92, pp. 1029–1057.

Patrenahalli et al., "A Branch and Bound Algorithm for Features Subset Selection", IEEE Transactions on Computers, vol. C–26, No. 9, 9/77, pp917–922.

Okada et al., "An Optimal Orthonormal System for Discriminant Analysis", Pattern Recognition, vol. 18, No. 1, 1985, pp. 139–144.

Riccia et al., "Fisher Discriminant Analysis and Factor Analysis", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI–5, No. 1, Jan. 1993, pp. 99–104.

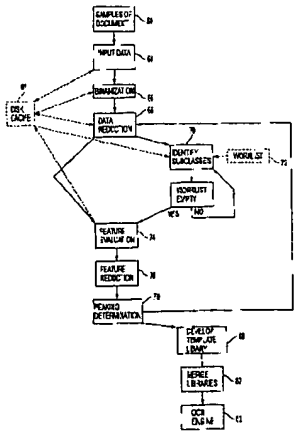
(List continued on next page.)

Primary Examiner—Samir Ahmed
(74) *Attorney, Agent, or Firm*—Venable LLP.; Andrew C. Aitken

(57) **ABSTRACT**

A computer automated feature selection method based upon the evaluation of hyper-rectangles and the ability of these rectangles to discriminate between classes. The boundaries of the hyper-rectangles are established upon a binary feature space where each bit indicates the relationship of a real feature value to a boundary within the minimum and maximum values for the feature across all samples. Data reduction combines the binary vector spaces so that the number of samples within a single class is within a range which is computationally feasible. Identification of subclasses identifies maximal subsets of S^+ which are exclusive against S^-. Feature evaluation determines within a single subclass the contribution of each feature towards the ability to discriminate the subclass from S^-. The base algorithm examines each feature, dropping any feature which does not contribute towards discrimination. A pair of statistics are generated for each remaining feature. The statistics represent a measure of how many samples from the class are within the subclass and a measure of how important each feature is to discriminating the subclass from S^-. The values for each subclass are then combined to generate a set of values for the class. These class feature metrics are further merged into metrics evaluating the features contribution across the entire set of classes. Feature reduction determines which features contribute the least across the entire set of classes.

18 Claims, 5 Drawing Sheets



OTHER PUBLICATIONS

Ronald et al., "Feature Induction by Backpropagation", IEEE International Conf. on Neural Networks, Jan. 1994, pp. 531–534.

Rounds et al., "A Combined Nonparametric Approach to Features Selection and Binary Decision Tree Design", Pattern Recognition, vol. 12, 1979, pp. 313–317.

Sethi et al., "Design of Multicategory Multifeature Split Decision Trees Using Perceptron Learning", Pattern Recognitiion, vol. 27, 1994, pp. 939–947.

Siddiqui et al., "Best Feature Selection Using Successive Elimination of Poor Performers", Proceedings of the Annual Conf. on Engineering in Med. & Biol., v. 15, Feb. 1993, pp. 725–726.

Solberg et al., "Automatic Feature Selection in Hyperspectral Satellite Imagery", IGARSS v 1 1993, pp. 472–475.

Sudhanva et al., "Dimensionality Reduction Using Geometric Projections: A New Technique", Pattern Recog., vol. 25 No. 8, 1992, pp. 809–817.

Wang et al., "Analysis and Design of a Decision Tree Based on Entropy Reduction and Its Application to Large Character Set Recognition", IEEE Trans. on Pattern Ana. & Mach. Intell., vol. PAMI–6, No. 4 1984, pp406–417.

Yu et al., "A More Efficient Branch and Bound Algorithm for Feature Selection", Pattern Recognition, vol. 26, No. 6, 1993, pp. 883–889.

Baim, "A Method for Attribute Selection in Inductive Learning Systems" IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 10, No. 6, Nov. 1988, pp. 888–896.

Chng et al., "Reducing the Computational Requirement of the Orthogonal Least Squares Algorithn" Proceedings—ICASSP, IEEE International Conference on Acoustics, v. 3 1994, pp. III–529–III–532.

Chng et al., "Backtracking Orthogonal Least Squares Algorithm for Model Selection" IEE Colloquium (Digest) n 034 Feb. 10, 1994, pp 10/1–10/6.

DiGesu et al., "Features Selection and 'Possiblity Theory'", Pattern Recognition, vol. 19, No. 1, 1986, pp. 63–72.

Fukunaga et al., "Nonparametric Discriminant Analysis" IEEE Transactions on Pattern & Machine Intelligence, vol. PAMI–5, No. 6, 11/83, pp671–678.

Gu et al., "Application of a Multilayer Decision Tree in Computer Recognition of Chinese Characters", IEEE Transactions on Pattern Analysis & Machine Intelligence, vol. PAMI–5, No. 1, 1/83, pp83–89.

Hamamoto et al., "On a Theoretical Comparison Between the Orthonormal Discriminant Vector Method and Discriminant Analysis", Pattern Recognition, Vo. 26, No. 12, 1993, pp 1863–1867.

Ichino et al., "Optimin Feature Selection by Zero–One Integer Programming" IEEE Trans. on Systems, Man & Cybernetics, vol. SMC–14, No. 5, 1984, pp737–746.

Kubichek et al., "Statistical Modeling and Feature Selection for Seismic Pattern Recognitiion", Pattern Recognition, vol. 18, No. 6, 1985, pp441–448.

Kudo et al., Optimal Subclasses with Dichotomous Variables for Feature Selection and Discrimination:, IEEE Transactions on Systems, Man and Cybernetics, vol. 19, No. 5, Sep./Oct. 1989, pp1194–1199.

Kudo et al., "Feature Selection Based on the Structural Indices of Categories", Pattern Recognition, vol. 26, No. 6, 1993, pp. 891–901.

Liu et al., "Algebraic Feature Extraction for Image Recognition Based on an Optimal Discriminant Criterion", Pattern Recognition, vol. 26, No. 6, pp. 903–911, 1993.

* cited by examiner

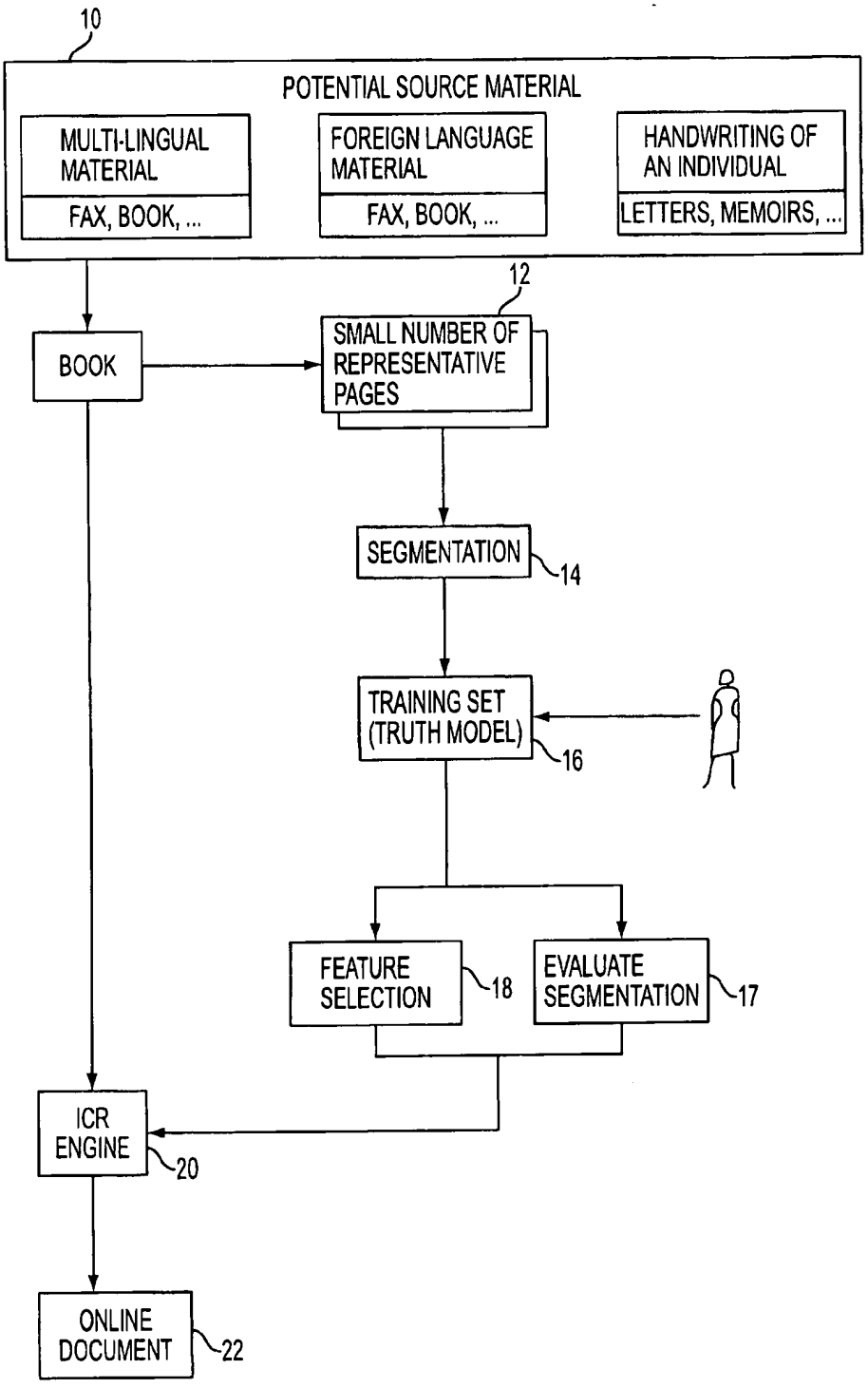

10
POTENTIAL SOURCE MATERIAL
MULTI-LINGUAL MATERIAL
FAX, BOOK, ...
FOREIGN LANGUAGE MATERIAL
FAX, BOOK, ...
HANDWRITING OF AN INDIVIDUAL
LETTERS, MEMOIRS, ...
12
BOOK
SMALL NUMBER OF REPRESENTATIVE PAGES
SEGMENTATION
14
TRAINING SET (TRUTH MODEL)
16
FEATURE SELECTION
18
EVALUATE SEGMENTATION
17
ICR ENGINE
20
ONLINE DOCUMENT
22

FIG. 1

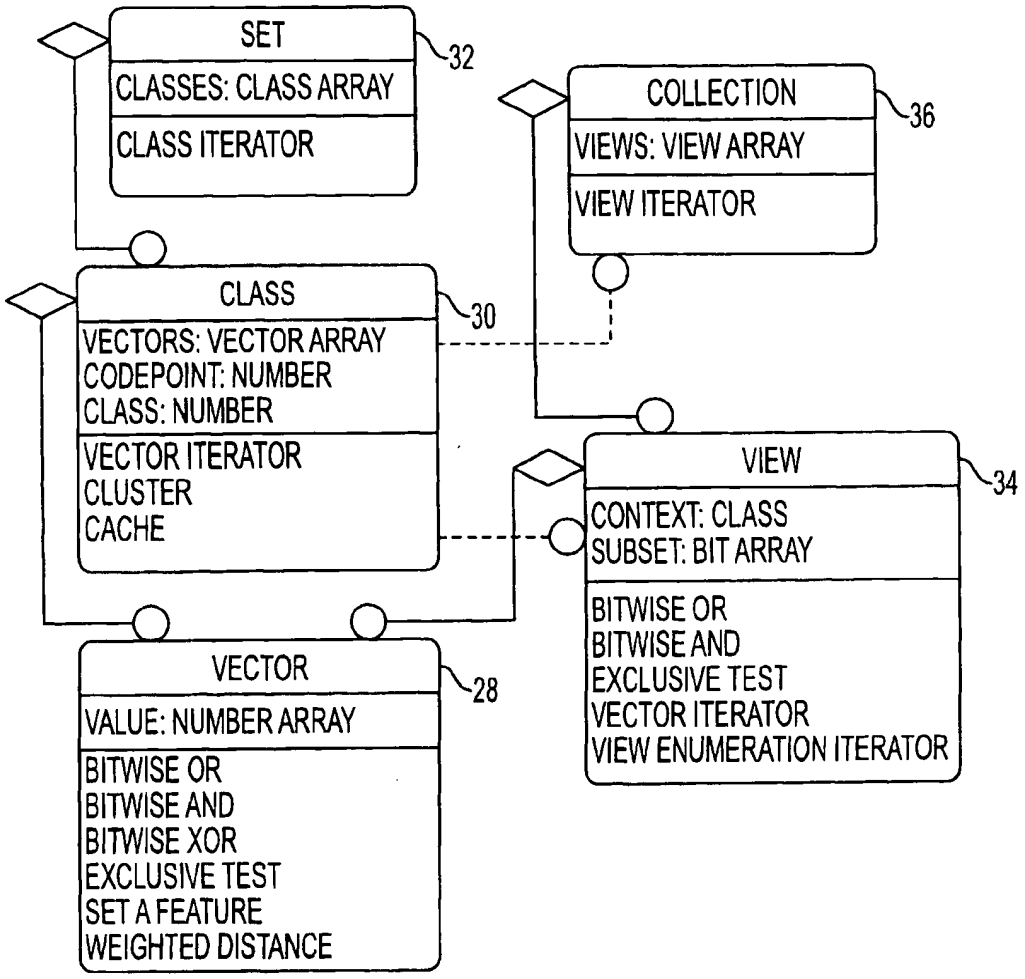

SET
CLASSES: CLASS ARRAY
CLASS ITERATOR
32
COLLECTION
VIEWS: VIEW ARRAY
VIEW ITERATOR
36
CLASS
VECTORS: VECTOR ARRAY
CODEPOINT: NUMBER
CLASS: NUMBER
VECTOR ITERATOR
CLUSTER
CACHE
30
VIEW
CONTEXT: CLASS
SUBSET: BIT ARRAY
BITWISE OR
BITWISE AND
EXCLUSIVE TEST
VECTOR ITERATOR
VIEW ENUMERATION ITERATOR
34
VECTOR
VALUE: NUMBER ARRAY
BITWISE OR
BITWISE AND
BITWISE XOR
EXCLUSIVE TEST
SET A FEATURE
WEIGHTED DISTANCE
28

FIG. 2

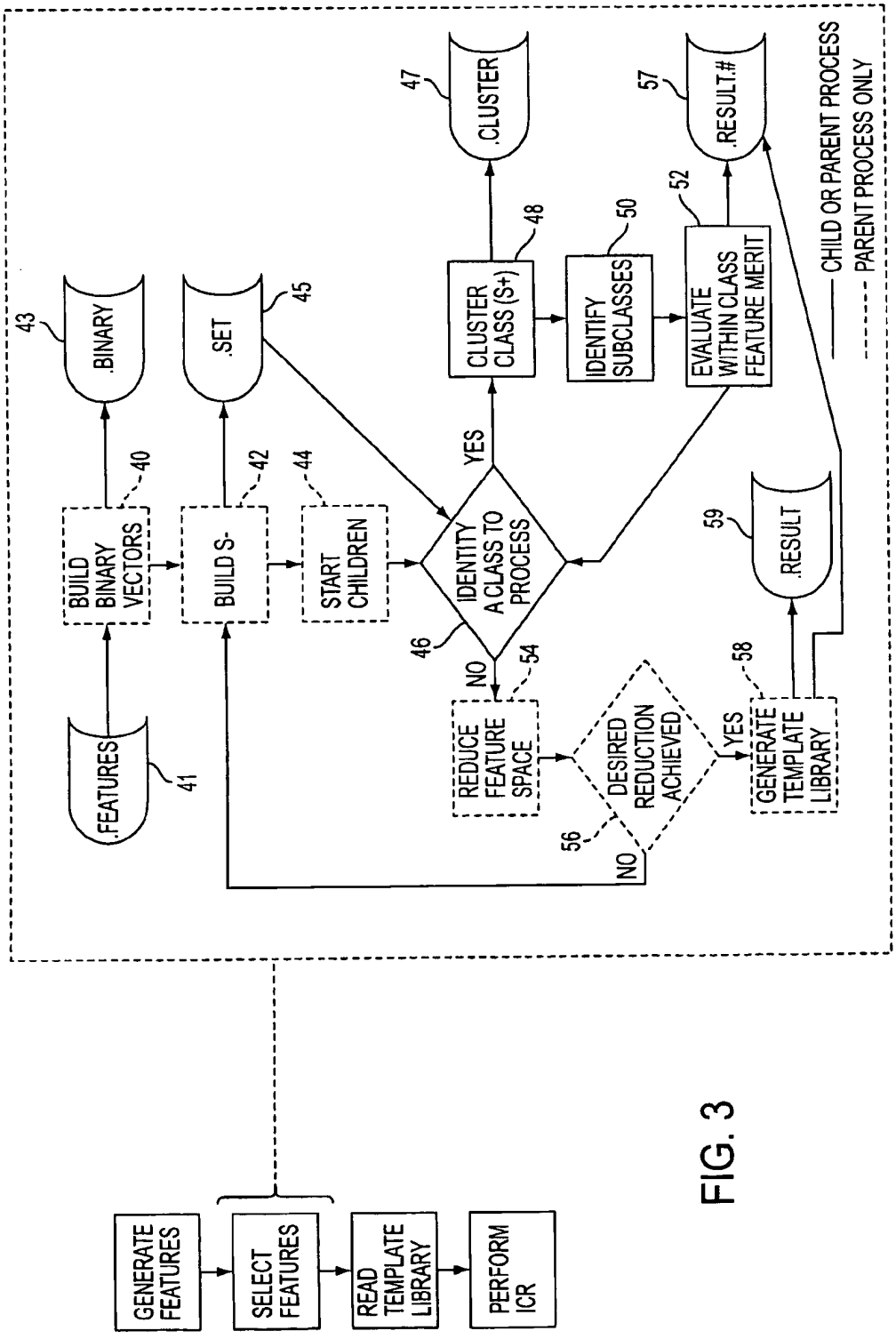

GENERATE FEATURES
SELECT FEATURES
READ TEMPLATE LIBRARY
PERFORM ICR
.FEATURES
41
BUILD BINARY VECTORS
40
.BINARY
43
BUILD S-
42
.SET
45
START CHILDREN
44
IDENTITY A CLASS TO PROCESS
46
YES
CLUSTER CLASS (S+)
48
.CLUSTER
47
IDENTIFY SUBCLASSES
50
EVALUATE WITHIN CLASS FEATURE MERIT
52
.RESULT#
57
NO
54
REDUCE FEATURE SPACE
DESIRED REDUCTION ACHIEVED
56
NO
YES
58
GENERATE TEMPLATE LIBRARY
.RESULT
59
CHILD OR PARENT PROCESS
PARENT PROCESS ONLY

FIG. 3

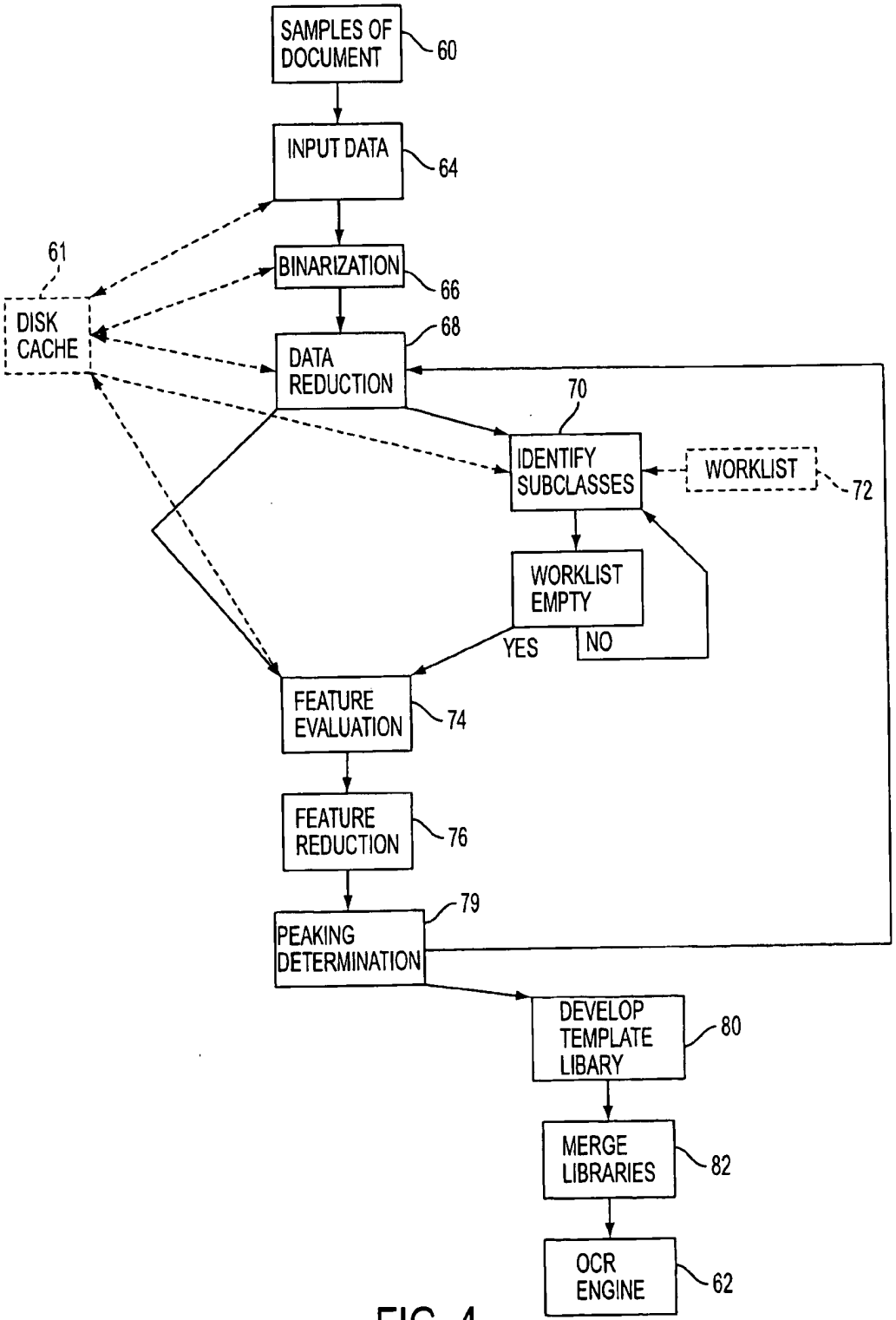

SAMPLES OF DOCUMENT
60
INPUT DATA
64
61
BINARIZATION
66
DISK CACHE
68
DATA REDUCTION
70
IDENTIFY SUBCLASSES
WORKLIST
72
WORKLIST EMPTY
YES
NO
FEATURE EVALUATION
74
FEATURE REDUCTION
76
79
PEAKING DETERMINATION
DEVELOP TEMPLATE LIBARY
80
MERGE LIBRARIES
82
OCR ENGINE
62

FIG. 4

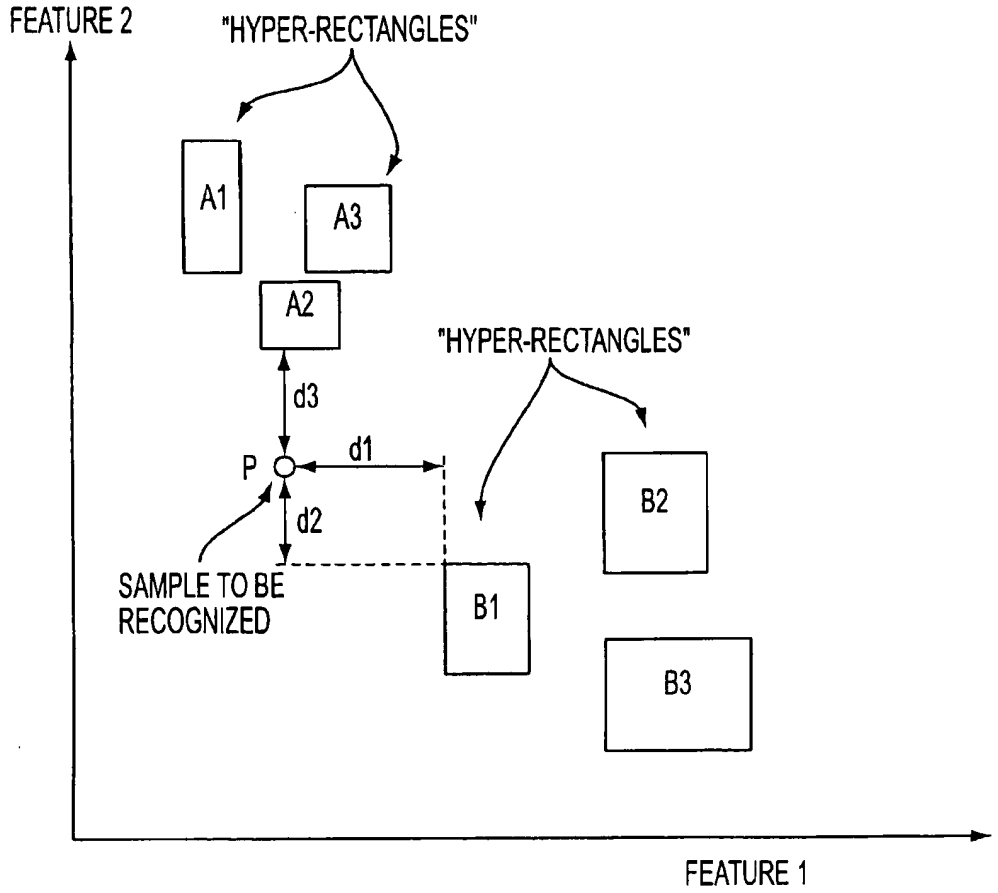

FEATURE 2
"HYPER-RECTANGLES"
A1
A3
A2
"HYPER-RECTANGLES"
d3
d1
P
d2
B2
SAMPLE TO BE RECOGNIZED
B1
B3
FEATURE 1

FIG. 5

AUTOMATIC METHOD FOR DEVELOPING CUSTOM ICR ENGINES

This invention was made with Government support under contract MDA-904-92-C-M300 awarded by the Department of Defense. The Government has certain rights in this invention.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to a computer automated method for creating image recognition engines optimized for a given language and/or source; and more particularly to a computationally efficient method for selecting, from a large universe of features, subsets of features that optimize recognition accuracy against a target data set. While not limited thereto, the invention will be explained in its preferred context for use in recognition of hand printed or machine printed characters.

2. Description of the Prior Art

As will be appreciated by those skilled in the art, each new language studied (and research in character recognition generally), proposes new character features to address specific problems or attributes of the source data set. However, the addition of new features is not necessarily sufficient for improved performance of a character recognition engine. If too many features are used, the rate of successful character discrimination actually drops as more features are added, a phenomena known in the art as peaking.

Each language (Japanese, Thai, English, et al.) has unique characteristics. Traditionally image character recognition (ICR) engines have been developed that make use of a feature set which was hand developed to support the language. The quality of the ICR is directly related to the amount of research effort applied to the language. High quality ICR engines exist where large markets support a large investment in research and development. An automated method that selects feature subsets would allow inexpensive development of ICR engines that perform well against languages that economically would not support a large investment.

Then, too, mixed languages can present unique problems as an OCR engine may need different features to distinguish between the two (or more) languages than the feature set used to best recognize either language individually. An automated feature selection tool would generate a feature set that is tailored to handle the particular mix of languages involved.

The following is a definition of certain of the terms used herein:

Class: All samples of a given codepoint (character) within the training data set.

Subclass: The set of samples within a single class which share some common attribute(s).

Codepoint: The representation to a machine of what people would recognize as a character.

Feature Vector: The set of measurements for the feature universe for a single sample ($\vec{v}$)

S^+: The class under consideration.

S^-: The set of all classes within the problem space other than the codepoint under consideration (S^+).

Exclusive: Used to describe a binary vector. Exclusive indicates that the binary vector is distinct from all vectors within S^- (eq. 1).

$$\vec{v} \Lambda \vec{S} \neq \vec{v} \forall \vec{S} \epsilon S^- \quad (1)$$

where:

Λ is the "and" operator,

$\forall$ is the "all" operator, and

ϵ means exists.

G: A subset of S^+, also known as a "view"

$\alpha(G)$: The binary vector resulting from the logical conjunction of the samples represented by G.

$\Theta(S^+,S^-)$: The collection of all G subsets in S^+ such that $\alpha(G)$ is exclusive against all vectors in S^-.

$\Omega(S^+,S^-)$: The collection of all maximal subsets in $\Theta(S^+,S^-)$ where maximal is defined as every H in $\Theta(S^+,S^-)$ such that if $\alpha(H)$ is not exclusive against some $\alpha(G)$ in $\Theta(S^+,S^-)$ then H=G.

MAX: Number of samples within the largest subclass occurring in $\Omega(S^+,S^-)$ (eq. 2).

$$\text{Max}(\Omega) = \max_{G \in \Omega} \frac{|G|}{|S^+|} \quad (2)$$

where: $|G|$ is the number of elements in set G.

AVE: The average number of samples within the subclasses occurring within $\Omega(S^+,S^-)$ (eq. 3).

$$Ave(\Omega) = \frac{1}{M} \sum_{G \in \Omega} \frac{|G|}{|S^+|} \text{ where } M = |\Omega| \quad (3)$$

SUMMARY OF THE INVENTION

An object of this invention is the provision of a computer automated method to select a subset of features in order to limit the number of features used by the character recognition engine, while optimizing its ability to discriminate among characters. That is, a method that narrows the number of features to those features that provide best within class consistency of recognition and the best cross-class discrimination.

Another object of the invention is the provision of a computer automated method of feature selection that is suitable for use with a large number of classes (i.e. 1000+ characters) and a large number of features (i.e. 100+ features).

A further object of this invention is the provision of a computer automated feature selection method in which the selection program is executed in a distributed operation on multiple processors.

Briefly, this invention contemplates the provision of a computer automated feature selection method based upon the evaluation of hyper-rectangles and the ability of these rectangles to discriminate between classes. The boundaries of the hyper-rectangles are established upon a binary feature space where each bit indicates the relationship of a real feature value to a boundary within the minimum and maximum values for the feature across all samples. Data reduction combines the binary vector spaces so that the number of samples within a single class is within a range which is computationally feasible. Identification of subclasses identifies maximal subsets of S^+ which are exclusive against S^-. Feature evaluation determines within a single subclass the contribution of each feature towards the ability to discriminate the subclass from S^-. The base algorithm examines each feature, dropping any feature which does not contribute towards discrimination. A pair of statistics are generated for each remaining feature. The statistics represent a measure of how many samples from the class are within the subclass

and a measure of how important each feature is to discriminating the subclass from S^-. The values for each subclass are then combined to generate a set of values for the class. These class feature metrics are further merged into metrics evaluating the features contribution across the entire set of classes. Feature reduction determines which features contribute the least across the entire set of classes. These features will be removed from consideration. The algorithm drops features if they fail to reach a predetermined significance level. If all features are found to be significant then the feature with the lowest contribution metric is discarded. Finally, peaking determination is used to determine if the process should be repeated against the reduced feature space. The peaking determination is done by examining the rate of change within the significance metrics.

The basic algorithm is set forth in two articles by M. Kudo and M. Shimbo: "Feature Selection Based on the Structural Indices of Categories," *Pattern Recognition* 26 (1993) 891–901, and "Optimal Subclasses with Dichotomous Variables for Feature Selection and Discrimination," *IEEE Trans. Syst. Man Cybern.* 19 (1989) 1194–1199, which are incorporated herein by reference.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, aspects and advantages will be better understood from the following detailed description of a preferred embodiment of the invention with reference to the drawings, in which:

FIG. 1 is a diagram of a generalized procedure to provide custom generation of an image character recognition engine in accordance with the teachings of this invention.

FIG. **2** is a diagram using object model notation to illustrate the feature vector subsets useful in the feature selection method in accordance with the invention.

FIG. **3** is a block diagram of the distributed processing of feature vectors in the feature selection process of the invention.

FIG. **4** is a flow diagram of one embodiment of the method steps for feature selection in accordance with the invention.

FIG. **5** is an example of determining the minimized distance measure in two-dimensional feature space in accordance with one aspect of the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT OF THE INVENTION

Referring now to FIG. **1**, with the feature selection algorithm of the invention and a suitable prior art ICR tool, it is now possible to develop and test ICR engines that are customized to a source data set such as indicated in the block **10**, Potential Source Material. A small portion of the source data is selected as a training set in block **12**. Using a suitable prior art base ICR tool, this training set is properly segmented and a truth model established for each of the characters, blocks **14** and **16**.

The segmentation takes as input the document image and produces as output a list of character coordinates from the image. The literature contains many methods for doing segmentation, many of which would be suitable here. For example, the segmenter used in the preferred embodiment used image histograms to determine the lines and then clustered the connected components in a line, broken into strokes where needed, into the characters.

The resulting characters are then automatically clustered into groups which is an attempt to pull out the different characters. An expert shown in silhouette then corrects any segmentation errors and/or grouping errors and tags the characters with the desired codepoints, block **17**.

As will be explained in more detail subsequently, at block **18**, real feature vectors are generated for the feature universe under examination. The real feature vectors are converted to binary vectors as part of the feature selection algorithm. The feature selection algorithm then processes each class to determine the maximal exclusive subsets and the corresponding contribution metrics. The feature selection algorithm continues to reduce the feature universe until a peaking determination is made. Once the final feature set is established, an ICR template is generated that corresponds to the input training data.

Once the template is prepared, the remaining source data can be processed by the ICR engine **20** as an online document, block **22**. This consists of reading the template library, segmenting the input data, and performing the recognition based upon the minimized distance measures.

The process may be repeated as often as necessary. Examination of alternative feature sets may be performed as new features are proposed by research efforts. The process would be repeated to generate new engines to support additional languages or data sources.

Referring now to FIG. **2**, it represents the data structure used in the automated feature selection process. The Figure uses Object Model Notation as described in the book by J. Rumbaugh et al. entitled *Object Oriented Modeling and Design*, 1991, Prentice Hall. The diamond shape symbol signifies "one" connected to the filled in circle which signifies "many." As seen in the figure, one set has many classes. The dotted lines indicate a conceptual relationship which is not necessarily required in the implementations. The fundamental unit of data is the binary feature "Vector" represented in block **28**. Binary feature vectors are aggregated into a "Class" block **30**, which represents a character, or one of several representations of a character (i.e. S^+). The classes are then aggregated into a "Set" (block **32**) which contains the entire universe of training data. A "View" (block **34**) is an aggregation of vectors within one class. And finally a "Collection" (block **36**) is an aggregation of views, which all correspond to the same class. Each of these has a set of data attributes and functions used to operate on the data attributes.

An implementation of the distributed, iterative feature evaluation tool is represented in FIG. **3**. Within the select features activity, processing consists of three phases; data initialization, processing of classes, and result gathering. The processing architecture is designed to support the use of multiple processes against the same set of data as well as the ability to interrupt and restart processing.

Data initialization consists of three activities; "Build binary vectors" (block **40**), "Build S^-" (block **42**), and "Start children" (block **44**). These activities are performed only by a parent process. The build binary vector step **40** consists of converting real feature vectors from a feature file **41** into binary feature vectors which are stored in a binary feature file **43**. The resultant binary feature vectors contain feature values for the entire feature universe (S^-), which are stored in a set file **45**. A mapping is maintained to identify the current feature set. As each class is binarized it is placed into the set, S^-. Since S^- contains every class, during processing it is necessary to skip the codepoint of the class under consideration (S^+) when it is encountered within S^-. If the number of samples in a class is larger than a user specified parameter, the class is reduced to a desired number of

samples. The reduction for S^- is based on Hamming distance and logical disjunction. The parent then initiates child processes (block **44**) for processing individual classes. The use of children and the number to use can be selected by the user.

In decision block **46**, the parent and children processes select classes for processing until each class has been processed using the current feature set (S^-). After selecting the class to be processed as S^+ (block **48** with clusters stored in cluster file **47**), the process determines if reduction in sample size is required. Reduction is recommended if more than 32 samples exist in a class (i.e. 32 samples allows a view to be manipulated using integer operations). The exact limit can be specified by the user. Once the class has been prepared for use as S^+, maximal exclusive subsets are identified using a non-recursive version of the base algorithm described in the Kudo et al. articles in place of the recursive version described in the articles, block **50**. For each of these maximal exclusive subsets, it is evaluated against the current feature set based upon the subset's ability to discriminate against S^-, block **52**. The binary vector which represents the view has all redundant features cleared. The view, the resulting vector from the view, and the feature evaluations are stored to memory.

Once each class has been processed against S^-, the parent process combines the metrics for each class so that a single metric is available that describes the contribution of each feature in the current feature set, block **52**. Those features which do not contribute significantly (as configured by the user) are discarded, block **54**. If all the features are significant, those that contribute the least will be removed. The user can configure a desired rate of feature removal. The structure indices are then tested to see if the feature evaluation process should terminate, block **56**. Once the final feature removal has occurred, the last set from result files **57** and **59** combine in generating the template library, block **58**.

Referring now to FIG. **4**, it shows how, in accordance with the teachings of this invention, the automated process for feature selection is incorporated into an automated method of creating an OCR engine for a given language and/or source. An optional disk cache **61** is used as a temporary storage in those hardware implementations where adequate common memory is not available. It starts with document samples (block **60**) and ends with an OCR engine **62** optimized for the sample source. As previously explained, the sample data features are inputted, block **64**, and binarized, block **66**. Binarization converts real feature vectors to binary feature vectors. To efficiently use the bit space allotted to the binary vector, thresholds are defined; e.g. (P+2), P is defined in the article by M. Kudo and M. Shimbo entitled "Feature Selection Based on the Structural Indices of Categories," Pattern Recognition 26 (1993), page 893, column 2, and the real sample vector is transformed based on its position relative to the thresholds.

A data reduction step **68** includes the Identify Subclass step of FIG. **3**. The activity "Identify Subclass," block **70**, requires the greatest amount of CPU time. The base algorithm for finding subclasses described in the M. Kudo and M. Shimbo article entitled "Optimal Subclasses with Dichotomous Variables for Feature Selection and Discrimination," IEEE Trans. Syst. Man Cybern., 19 (1989) pp. 1194–1199, performs at least one iteration through the recursive procedure ENUMSAT. The class S^+ may not contain any subclasses. Predetermination of the existence of subclasses within S^+ can completely remove the "Identify Subclass" activity. The existence of subclasses is easy to detect. If a vector generated from the view consisting of each sample in S^+ is exclusive against S^-, then each sample is exclusive against S^-. In this case a single view consisting of each sample will be the only entry within the collection $\Omega(S^+,S^-)$.

To reduce the total number of iterations required, once a subclass divides, each resulting portion needs to be examined to determine if it is sufficient or if further subdivision is required. The same subdivision may be identified many times upon examination of the divisions. This duplication is redundant and in a large problem space requires significant processing time. To address this issue, an iterative algorithm is used in accordance with the teaching of this invention. A work list **72** is maintained that corresponds to the original recursive call. Redundant entries are not placed onto the worklist. This reduces significantly the number of traversals of S^- required to determine if subsets were exclusive.

The loop within the flow diagram (FIG. **3**) from the activity "Build S^-" through the decision "Desired Reduction Achieved" reduces the feature universe, block **74**. The base algorithm drops features from the universe using two rules. First, features are removed if the feature contribution does not exceed a configurable threshold, block **76**. Then, if no features were dropped in the first step a single feature is selected for removal. At block **79**, a peaking determination is used to determine if the process should be used against the reduced feature space. This peaking determination can be accomplished by examining the rate of change within the significant metrics.

The results from the feature selection mechanism are used to generate a template library, block **80**. As described in "Feature Selection Based on the Structural Indices of Categories", page 896:

c=a Class

$Sc^+=S^+$ for class c

$|Sc^+|$=number of sample vectors in Sc^+

$Sc^-=S^-$ for class c

$|Sc^-|$=number of sample vectors in Sc^-

G=a view in $\Omega(Sc^+, Sc^-)$

|G|=number of sample vectors in G

$\alpha(G,I)=\alpha(G)$ with the ith feature zeroed out (conceptually this removes both sides of the hyper-rectangle for that feature's dimension)

$C^-(\alpha(G,i))$=the subset of vectors in $\alpha(G,i)$ exclusive against Sc^-

$|C^-(\alpha(G,i))|$=number of sample vectors in $C^-(\alpha(G,i))$

$$p^+(G,c)=|G|/|Sc^+| \tag{4}$$

Conceptually eq. 4 is the degree of contribution of G to Sc^+

$$p^-(G,i,c)=|C^-(\alpha(G,i)|/|Sc^-| \tag{5}$$

Conceptually eq. 5 is the degree to which feature I is important in order to make G be exclusive against Sc^-.

Now the contribution metric for a feature I for class c is the summation across all G in $\Omega(Sc^+, Sc^-)$ of $[p^+(G,c)*p^-(G,i,c)]$.

The contribution metric for a feature across the entire Set (all classes) is the summation of all contribution metrics for a feature across all classes.

Note however that features are left zeroed out if the contribution metric for the feature is zero (i.e. $|C^-(\alpha(G,i))|$ is zero). This implies that feature confidence metrics are dependent on previous iterations. Using one pass through the features for one G would tend to favor the features looked at last. In the present invention to avoid favoring the features looked at last, the following steps are followed:

1) Calculate the contribution metrics as before.

2) Sort the features from the largest contribution to least.

3) Reset all of the features to their original values.

4) Pass through the features as before using the new order.

5) Repeat, starting at step 2, unless one of the following conditions are met:
   a) A maximum number of iterations allowed is exceeded (on the order of 5).
   b) No new features are added to the set of good features.
   c) The contribution metrics remain substantially unchanged as the evaluation order changes.

Contribution metrics will change when evaluated in a different order. When no significant change occurs as indicated in condition c above it is an indication that the process can stop. The template library **80** is then used by the recognition engine to process source samples.

The parallel implementation is not graphically represented here. Basically, the steps between and including data reduction and feature evaluation can be done independently for each S^+ and the resulting template libraries merged in step **82**.

Given the feature selection algorithm and a base ICR tool, it is now possible to develop and test ICR engines that are customized to the source data set. A small portion of the source data is selected as the training set. Using the base ICR tool this training set is properly segmented and a truth model established for each of the characters. Real feature vectors are generated for the feature universe under examination.

The real feature vectors are converted to binary vectors within the feature selection algorithm. The feature selection algorithm then processes each class to determine the maximal exclusive subsets and the corresponding contribution metrics. The feature selection algorithm continues to reduce the feature universe until a peaking determination is made. Once the final feature set is established, the ICR template is generated that corresponds to the input training data.

Once the template is prepared, the remaining source data can be processed by the ICR engine. This consists of reading the template library, segmenting the input data, and performing the recognition based upon the minimized distance measures.

The process may be repeated as often as necessary. Examination of alternative feature sets may be performed as new features are proposed by research efforts. The process would be repeated to generate new engines to support additional languages or data sources.

The results from the feature selection mechanism are used to generate a template library as set forth in the I. Bella and G. Macey paper "Feature Selection Given Large Numbers of Classes and a Large Feature Universe" Proceeding 1995 Symposium on Document Image Understanding Technology, October 24–25, pp 202–212. The paper is hereby incorporated by reference. This template library is then used by the recognition engine to process source samples. The data contained within the results and the comparison process is provided for completeness.

The feature extraction algorithm produces the following data:

- Thresholds (Bucket Ranges): The information required to convert from a real feature value to the binary representation.
- Feature Map: Identification of the features that comprise the final selected feature set.
- Class Data: there are multiple data elements generated for each class. The complete set is: the codepoint of the class, the binary vectors and the related confidence values representing the maximal exclusive subsets ($\Omega(S^+,S^-)$) within the class, and a final set of confidence values for the class as a whole. The class wide confidence values are aggregations of the confidence values for each subset.
- Final Confidence Values: A contribution metric for each feature in the result set. This metric is the aggregation of the class contribution metrics.

This set of data is used to recognize each character within the source materials.

The recognition of a source character is a three step process. The character image is converted into a binary feature vector using the thresholds. This binary vector is then compared against each of the template vectors generating a distance measure. Finally, the template which minimizes the distance measure is selected as the correct class for the source character.

The distance measured between a source binary vector (S) and template vector (T) each composed of N features is given by the two equations 6 and 7 below.

$$\vec{d}_i=(\vec{S}_i \vee \vec{T}_i)\wedge \vec{T}_i \tag{6}$$

$$D=\vec{d}\times\vec{C}=\sum_{1}^{N}(\vec{C}_i * \vec{d}_i) \tag{7}$$

Each bit in a binary vector represents a rule denoting whether the value for the examined feature is greater or less than a threshold or bucket edge. Each feature (i) produces a feature distance measure ($\vec{d}_i$) in equation 6 that represents how far the sample is from the edge of a template's valid range which is the interior of the hyper-rectangle. The final distance is the feature distance multiplied by the confidence value for that feature (C_i). The final distance function (D) may be used with any of the three different confidence vectors: subset, class, and set.

An example in two-dimensional feature space is shown in FIG. **5**. The distance from sample P (the sample to be recognized) to hyper-rectangle B1 is d1*c1(B1)+d2*c2(B1) where c1(B1) is the contribution factor of Feature **1** for B1 and likewise for c2(B1). The same distance is calculated between the sample P and all hyper-rectangles. A K-nearest neighbor algorithm is used to decide which hyper-rectangle wins. For K=1, this is simply the hyper-rectangle with the smallest distance. In this example, with K=1, the hyper-rectangle A3 might win with a distance of d3*c2(A2+0*c1 (A2) and hence the sample P would be classified as an 'A'.

While the invention has been described in terms of a single preferred embodiment, those skilled in the art will recognize that the invention can be practiced with modification within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is as follows:

1. A computer automated method for machine recognition of character images in source material including the steps of:

selecting a sample portion of said source material as a training set;

segmenting said training set;

grouping characters segmented in said segmenting step into classes;

generating feature vectors for each of said classes;

generating a feature set by processing each class to determine a maximal exclusive subset and a corresponding metric iteratively until a peaking determination is made, wherein said processing includes predetermining the existence of classes of said feature set by testing a vector generated from a view consisting of each sample in said feature set to determine if the vector is exclusive against said feature sets for all characters of said training set;

generating an image character recognition template corresponding to said training set;

processing on line said source material with character recognition template.

2. A computer automated method for machine recognition of character images in source material as in claim 1 including the further step of eliminating feature vectors that contribute less than a predetermined level of exclusivity of said feature set.

3. A computer automated method for machine recognition of character images in source material as in claim 1 wherein said generating step includes the further steps of maintaining a work list of subclasses and entering into said work list only subclasses not previously entered.

4. A computer automated method for creating an image recognition engine for a universe of characters comprising the steps of:

selecting samples from a universe of characters as a training set;

segmenting samples from the universe of characters;

determining a feature set for identifying each character in the samples;

evaluating features in said feature set to determine maximal subsets of said feature set that are exclusive of feature sets for all characters of said universe of characters, wherein said evaluating step includes a step of predetermining the existence of classes of said feature set by testing a vector generated from a view consisting of each sample in said feature set to determine if the vector is exclusive against said feature sets for all characters of said universe of characters;

determining the contribution metrics for each feature in each class for each set, wherein the classes are evaluated iteratively until a peaking determination is made;

eliminating features that contribute less than a predetermined level of exclusivity of said feature set; and

developing a template library of feature sets for use in an optical character recognition engine.

5. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 further comprising the steps of maintaining a work list of subclasses and entering into said work list only subclasses not previously entered.

6. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 including the further step of converting real feature vectors to binary feature vectors by defining a series of threshold values with a binary value assigned to each threshold value, and assigning the binary value to each real feature value based on its threshold value.

7. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 wherein said steps of evaluating and eliminating are repeated until a desired reduction in the feature universe is achieved and wherein a percentage of features are eliminated each time said steps of evaluating and eliminating are repeated.

8. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 including the further step of converting real feature vectors to binary feature vectors by defining a series of threshold values with a binary value assigned to each threshold value, and assigning the binary value to each real feature value based on its threshold value.

9. A computer automated method for creating an image recognition engine for a universe of characters as in claim 5 including the further step of converting real feature vectors to binary feature vectors by defining a series of threshold values with a binary value assigned to each threshold value, and assigning the binary value to each real feature value based on its threshold value.

10. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 wherein said steps of evaluating and eliminating are repeated until a desired reduction in the feature universe is achieved.

11. A computer automated method for creating an image recognition engine for a universe of characters as in claim 5 wherein said steps of evaluating and eliminating are repeated until a desired reduction in the feature universe is achieved.

12. A computer automated method for creating an image recognition engine for a universe of characters as in claim 6 wherein said steps of evaluating and eliminating are repeated until a desired reduction in the feature universe is achieved.

13. A computer automated method for creating an image recognition engine for a universe of characters as in claim 10 wherein a certain percentage of features are eliminated each time said steps of evaluating and eliminating are repeated.

14. A computer automated method for creating an image recognition engine for a universe of characters as in claim 11 wherein a certain percentage of features are eliminated each time said steps of evaluating and eliminating are repeated.

15. A computer automated method for creating an image recognition engine for a universe of characters as in claim 12 wherein a certain percentage of features are eliminated each time said steps of evaluating and eliminating are repeated.

16. A computer automated method for creating an image recognition engine for a universe of characters as in claim 4 wherein said step of evaluating features is carried out in parallel in a plurality of processors for a plurality of feature sets for identifying each character.

17. A computer automated method for creating an image recognition engine for a universe of characters by selecting, from a large universe of features, subsets of features to optimize recognition accuracy, comprising the steps of:

segmenting samples from the universe of characters;

extracting feature sets for each character in the sample;

determining binary vectors for each character from the extracted feature sets;

evaluating features in said feature set to determine maximal subsets of said feature set that are exclusive of feature sets for all characters of said universe of characters, wherein said evaluating step includes a step of predetermining the existence of classes of said feature set by testing a vector generated from a view consisting of each sample in said feature set to determine if the vector is exclusive against said feature sets for all characters of said universe of characters;

determining the contribution metrics for each feature in each class for each set, wherein the classes are evaluated iteratively until a peaking determination is made;

eliminating features that contribute less than a predetermined level to exclusivity of said feature set; and

developing a template library of feature sets for use in an optical character recognition engine.

18. The computer automated method of claim 4, further comprising the steps of:

determining binary vectors from the template library;

minimizing the distance measure between templates and the binary vectors by finding the distance between the binary vector, which is a point, and the closest template edge for each feature; and

providing each character associated with the closest template or templates as performed by the K-Nearest Neighbor test.

* * * * *

US005418888A

United States Patent [19]

Alden

[11] Patent Number: 5,418,888

[45] Date of Patent: May 23, 1995

[54] **SYSTEM FOR REVELANCE CRITERIA MANAGEMENT OF ACTIONS AND VALUES IN A RETE NETWORK**

[76] Inventor: **John L. Alden**, 7258-91st SE., Mercer Island, Wash. 98040

[21] Appl. No.: **894,353**

[22] Filed: **Jun. 4, 1992**

[51] **Int. Cl.**[6] **G06F 15/18**

[52] **U.S. Cl.** **395/64**; 395/600; 395/50; 395/52; 395/63; 364/DIG. 1; 364/274; 364/274.2

[58] **Field of Search** 395/64, 600, 50, 52, 395/63

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,849,905	7/1989	Loeb et al.	395/64
5,072,405	12/1991	Ramakrisna et al.	395/64
5,119,470	6/1992	Highland et al.	395/64

Primary Examiner—Thomas G. Black

Assistant Examiner—Wayne Amsbury

Attorney, Agent, or Firm—Graybeal Jackson Haley & Johnson

[57] **ABSTRACT**

A method for programming a computer, and software products for implementing the method, are disclosed. According to the method, objects are created which, upon execution of the system, may initiate actions and may acquire values. A specification of relevance criteria is stored as an attribute of each object. Relevance criteria, which may refer to the values of other objects, describe all conditions necessary and sufficient to make the binary decision for whether the object will be considered relevant to the program's operation during execution. Action criteria may also be specified to control the actions of a relevant object. The method is particularly well suited for programming expert systems, but may be used for programming any type of computer program.

22 Claims, 11 Drawing Sheets

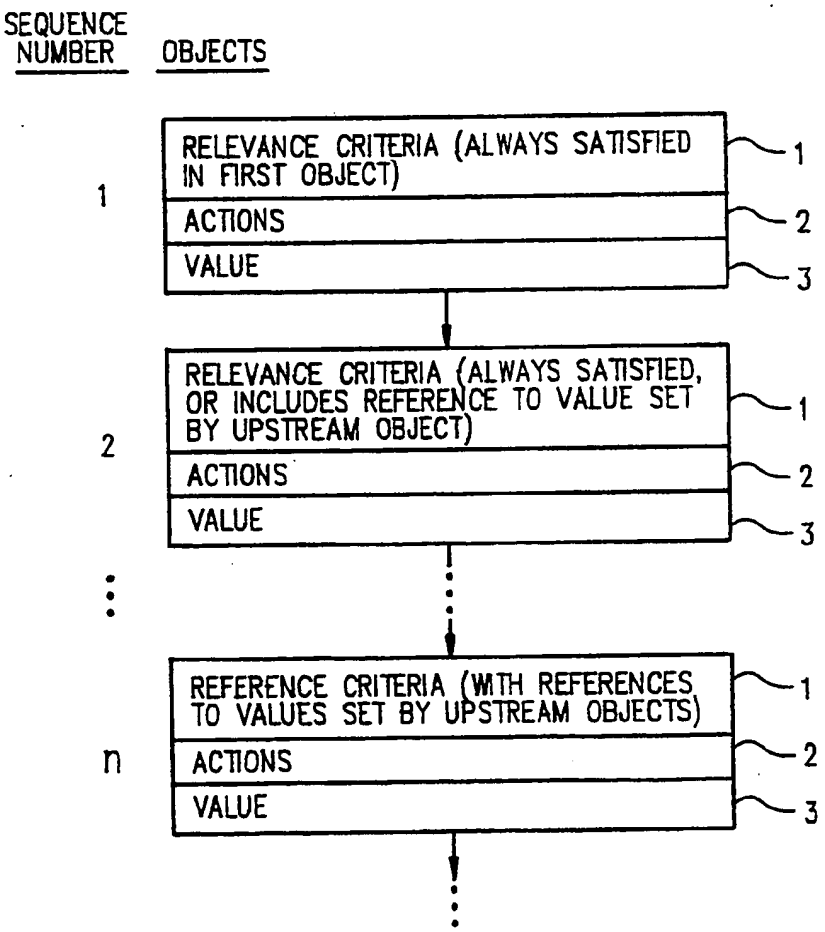


FIG. 1

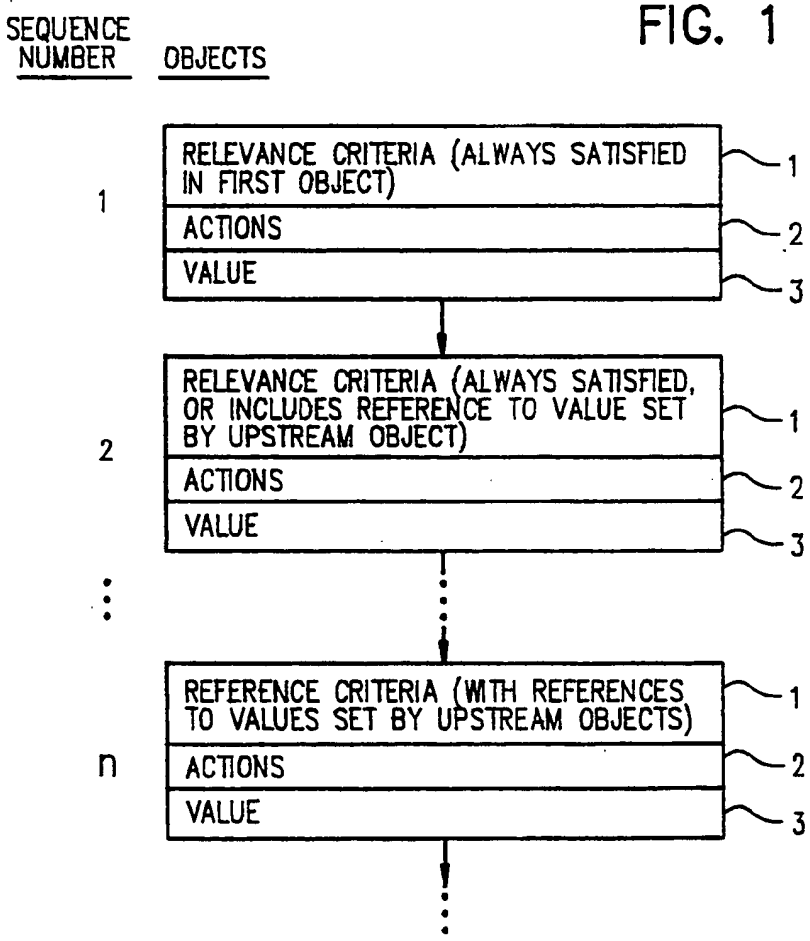

SEQUENCE NUMBER
OBJECTS
1
RELEVANCE CRITERIA (ALWAYS SATISFIED IN FIRST OBJECT)
ACTIONS
VALUE
2
RELEVANCE CRITERIA (ALWAYS SATISFIED, OR INCLUDES REFERENCE TO VALUE SET BY UPSTREAM OBJECT)
ACTIONS
VALUE
n
REFERENCE CRITERIA (WITH REFERENCES TO VALUES SET BY UPSTREAM OBJECTS)
ACTIONS
VALUE
1
2
3

FIG. 2

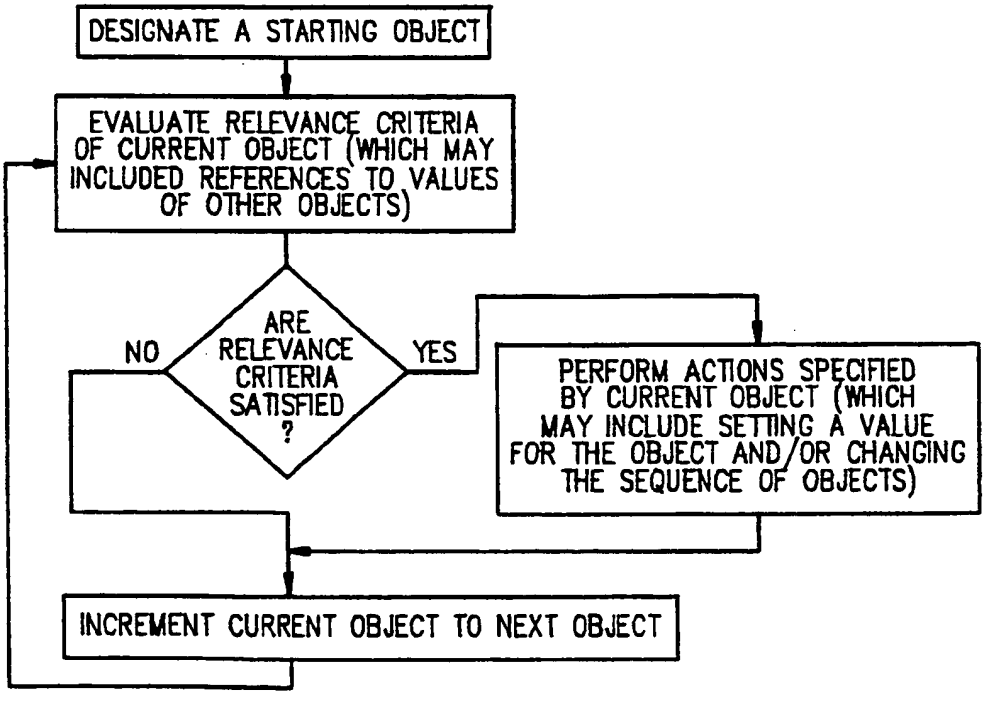
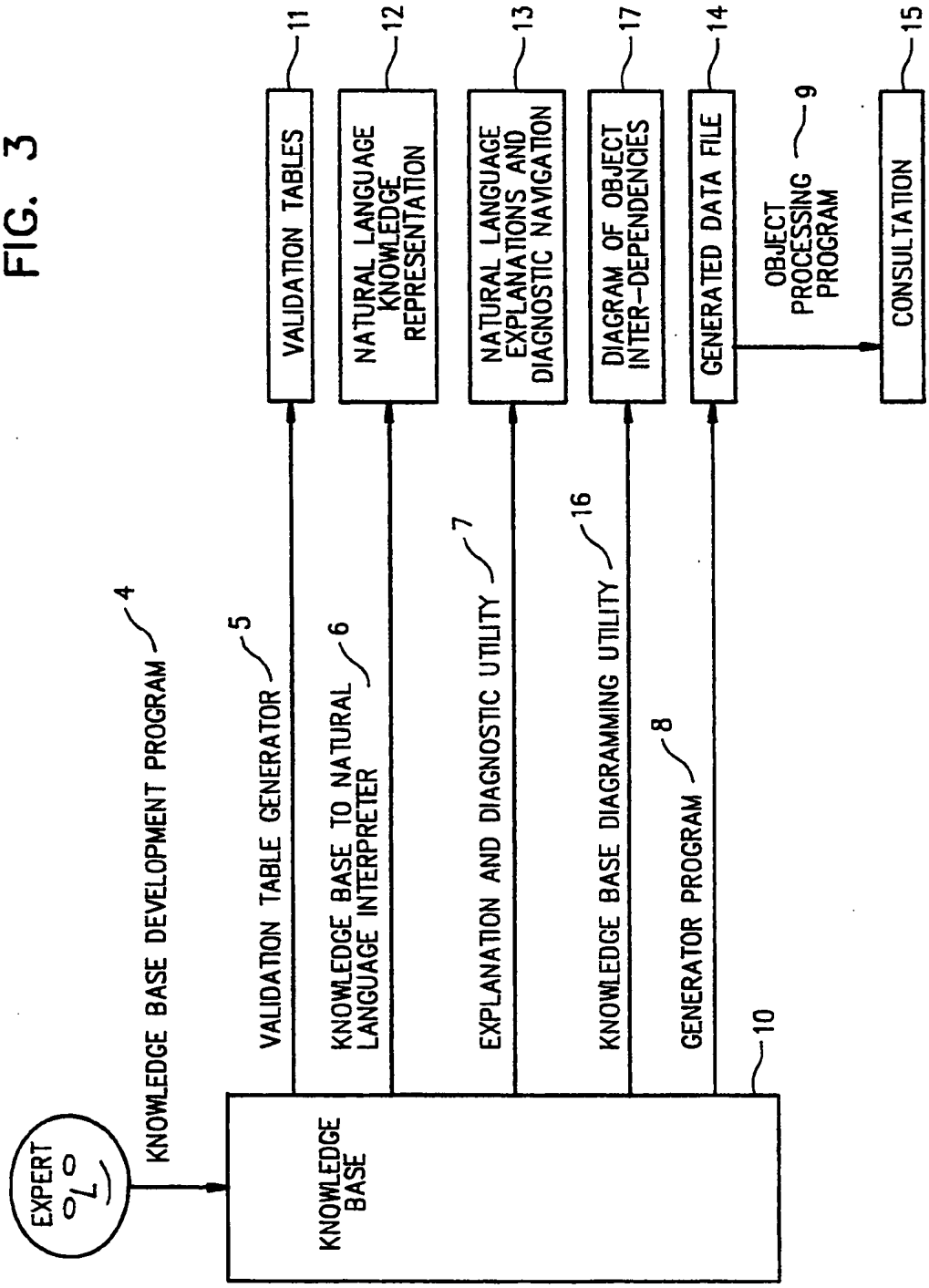

DESIGNATE A STARTING OBJECT
EVALUATE RELEVANCE CRITERIA OF CURRENT OBJECT (WHICH MAY INCLUDED REFERENCES TO VALUES OF OTHER OBJECTS)
ARE RELEVANCE CRITERIA SATISFIED ?
NO
YES
PERFORM ACTIONS SPECIFIED BY CURRENT OBJECT (WHICH MAY INCLUDE SETTING A VALUE FOR THE OBJECT AND/OR CHANGING THE SEQUENCE OF OBJECTS)
INCREMENT CURRENT OBJECT TO NEXT OBJECT

FIG. 3


EXPERT
10
KNOWLEDGE BASE DEVELOPMENT PROGRAM
4
KNOWLEDGE BASE
VALIDATION TABLE GENERATOR
5
KNOWLEDGE BASE TO NATURAL LANGUAGE INTERPRETER
6
EXPLANATION AND DIAGNOSTIC UTILITY
7
KNOWLEDGE BASE DIAGRAMMING UTILITY
16
GENERATOR PROGRAM
8
10
VALIDATION TABLES
11
NATURAL LANGUAGE KNOWLEDGE REPRESENTATION
12
NATURAL LANGUAGE EXPLANATIONS AND DIAGNOSTIC NAVIGATION
13
DIAGRAM OF OBJECT INTER-DEPENDENCIES
17
GENERATED DATA FILE
14
OBJECT PROCESSING PROGRAM
9
CONSULTATION
15

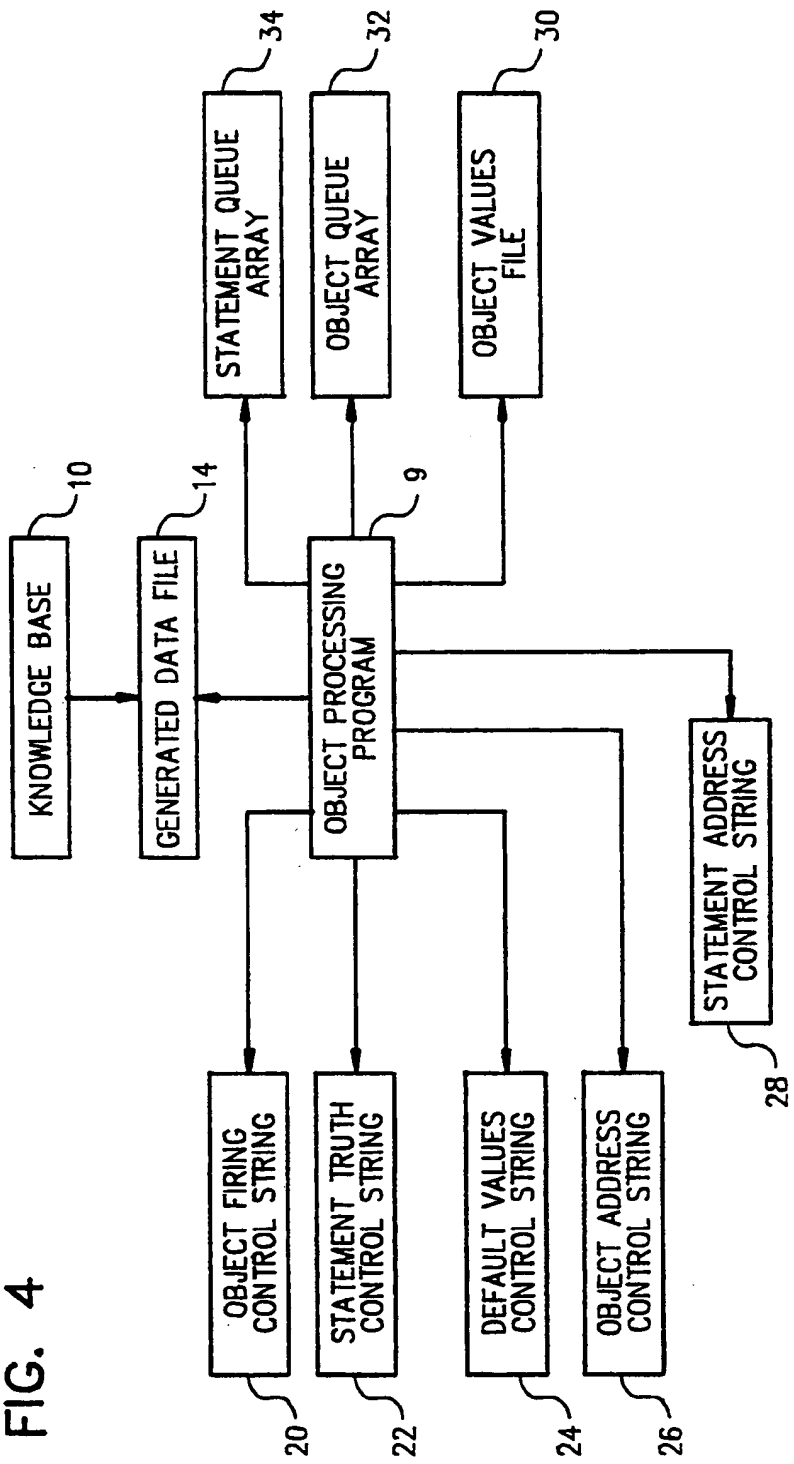

KNOWLEDGE BASE
10
GENERATED DATA FILE
14
OBJECT PROCESSING PROGRAM
9
STATEMENT QUEUE ARRAY
34
OBJECT QUEUE ARRAY
32
OBJECT VALUES FILE
30
STATEMENT ADDRESS CONTROL STRING
28
OBJECT FIRING CONTROL STRING
20
STATEMENT TRUTH CONTROL STRING
22
DEFAULT VALUES CONTROL STRING
24
OBJECT ADDRESS CONTROL STRING
26

FIG. 4

FIG. 5

KNOWLEDGE BASS STRUCTURE

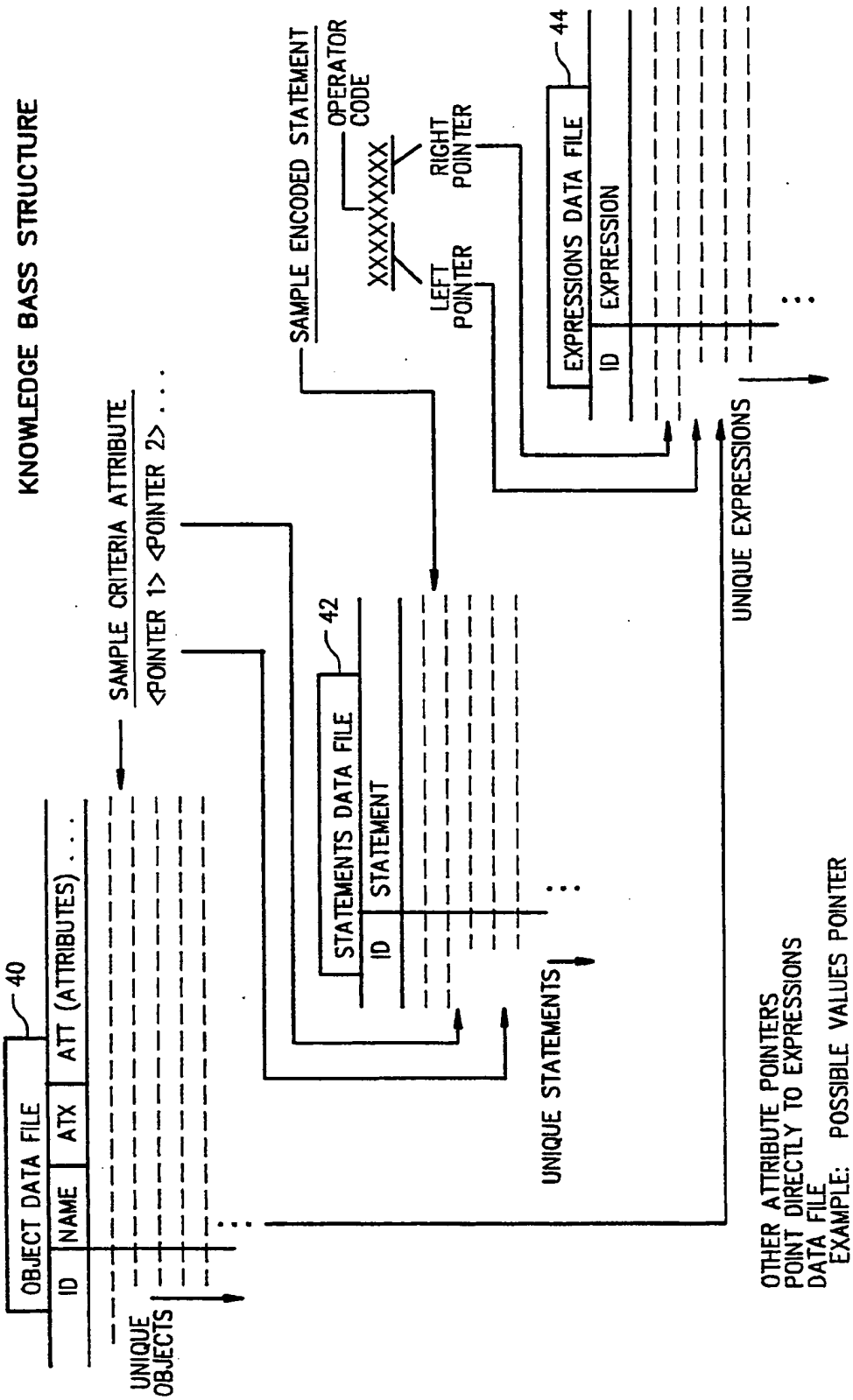


FIG. 6

GENERATED DATA FILE STRUCTURE

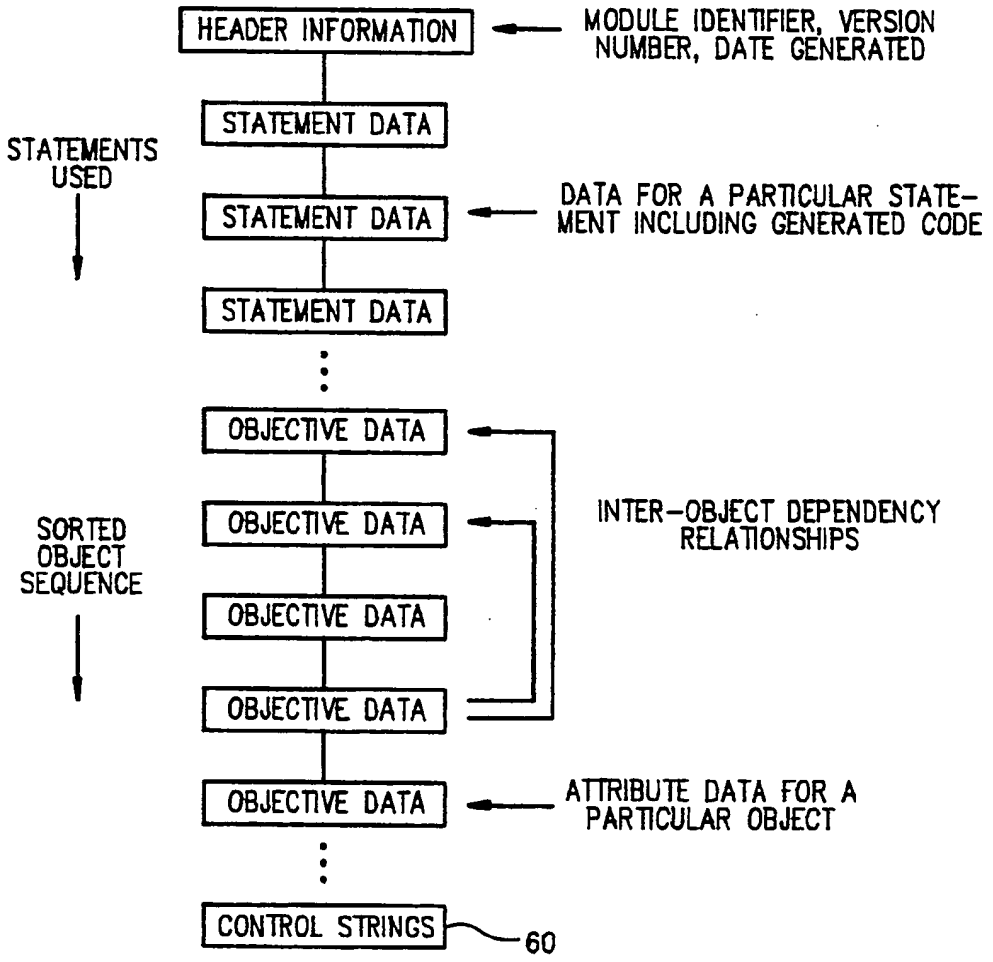


FIG. 7

CONTROL STRINGS STRUCTURE

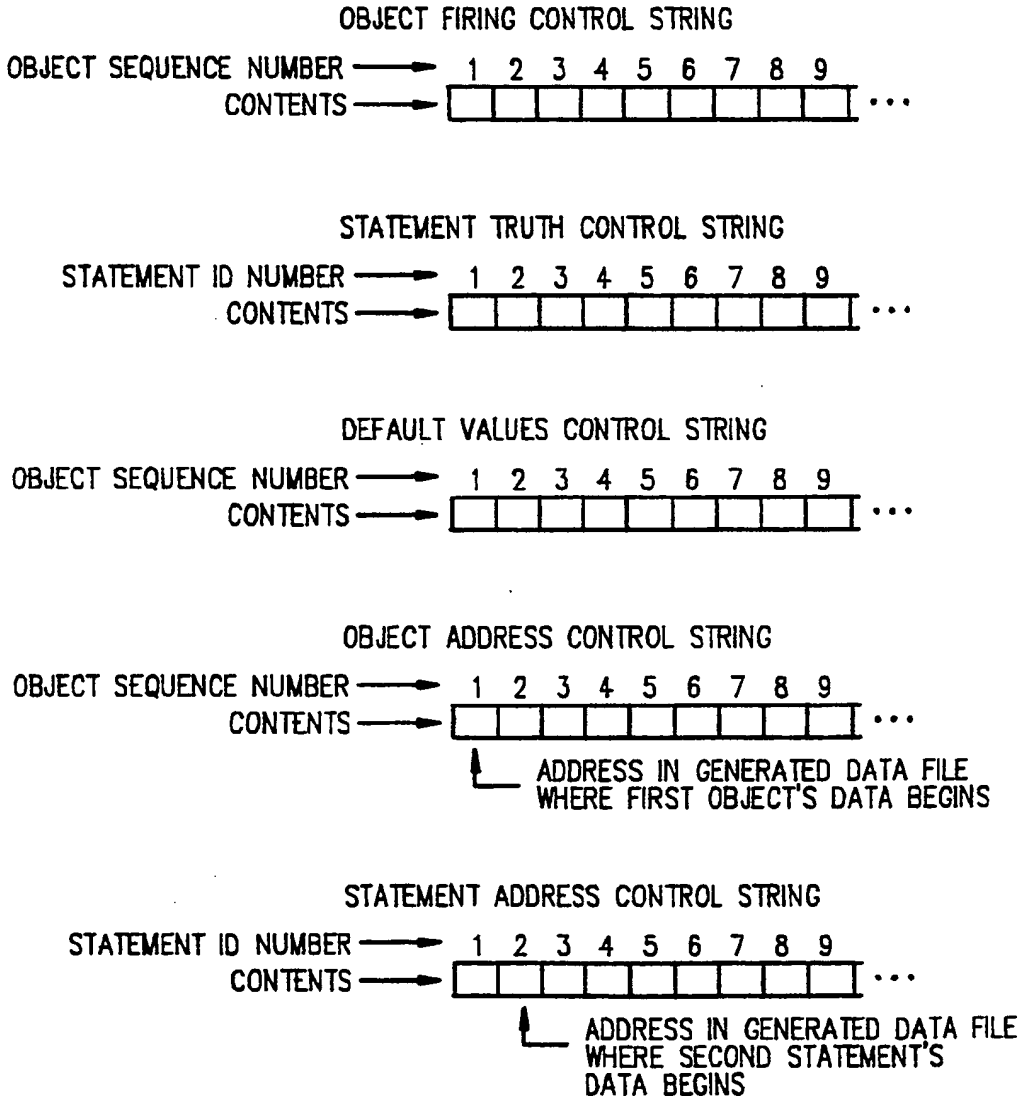


FIG. 8

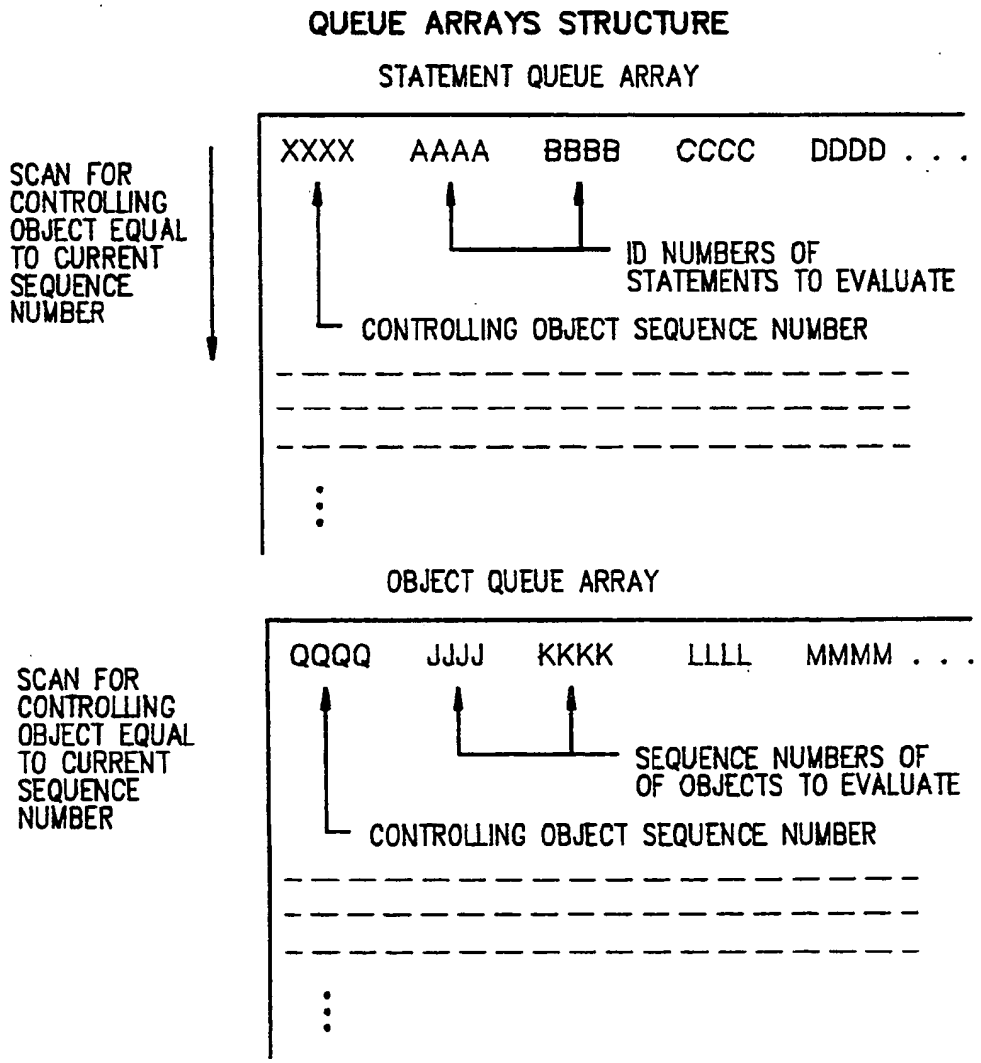

QUEUE ARRAYS STRUCTURE
STATEMENT QUEUE ARRAY
SCAN FOR CONTROLLING OBJECT EQUAL TO CURRENT SEQUENCE NUMBER
XXXX AAAA BBBB CCCC DDDD . . .
ID NUMBERS OF STATEMENTS TO EVALUATE
CONTROLLING OBJECT SEQUENCE NUMBER
OBJECT QUEUE ARRAY
SCAN FOR CONTROLLING OBJECT EQUAL TO CURRENT SEQUENCE NUMBER
QQQQ JJJJ KKKK LLLL MMMM . . .
SEQUENCE NUMBERS OF OF OBJECTS TO EVALUATE
CONTROLLING OBJECT SEQUENCE NUMBER

FIG. 9

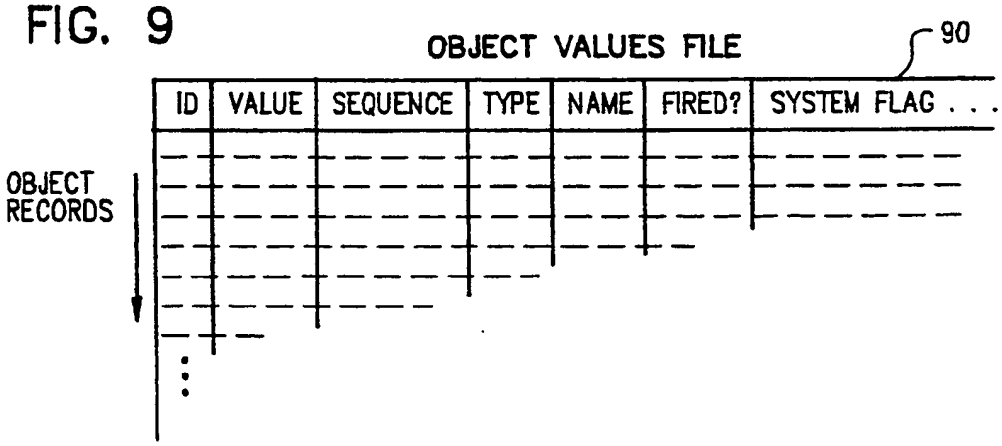

OBJECT VALUES FILE
90
ID
VALUE
SEQUENCE
TYPE
NAME
FIRED?
SYSTEM FLAG . . .
OBJECT RECORDS

FIG. 10

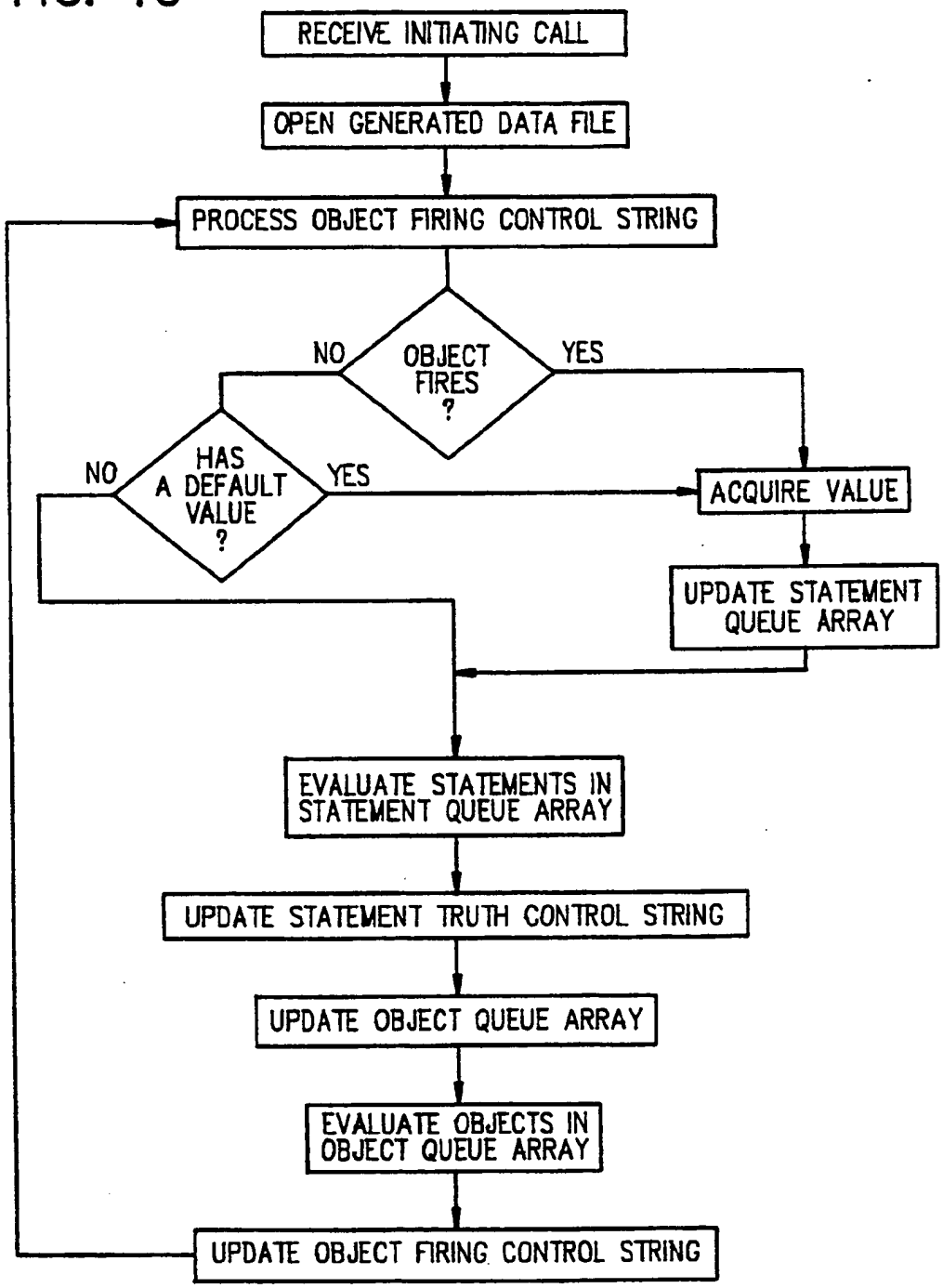
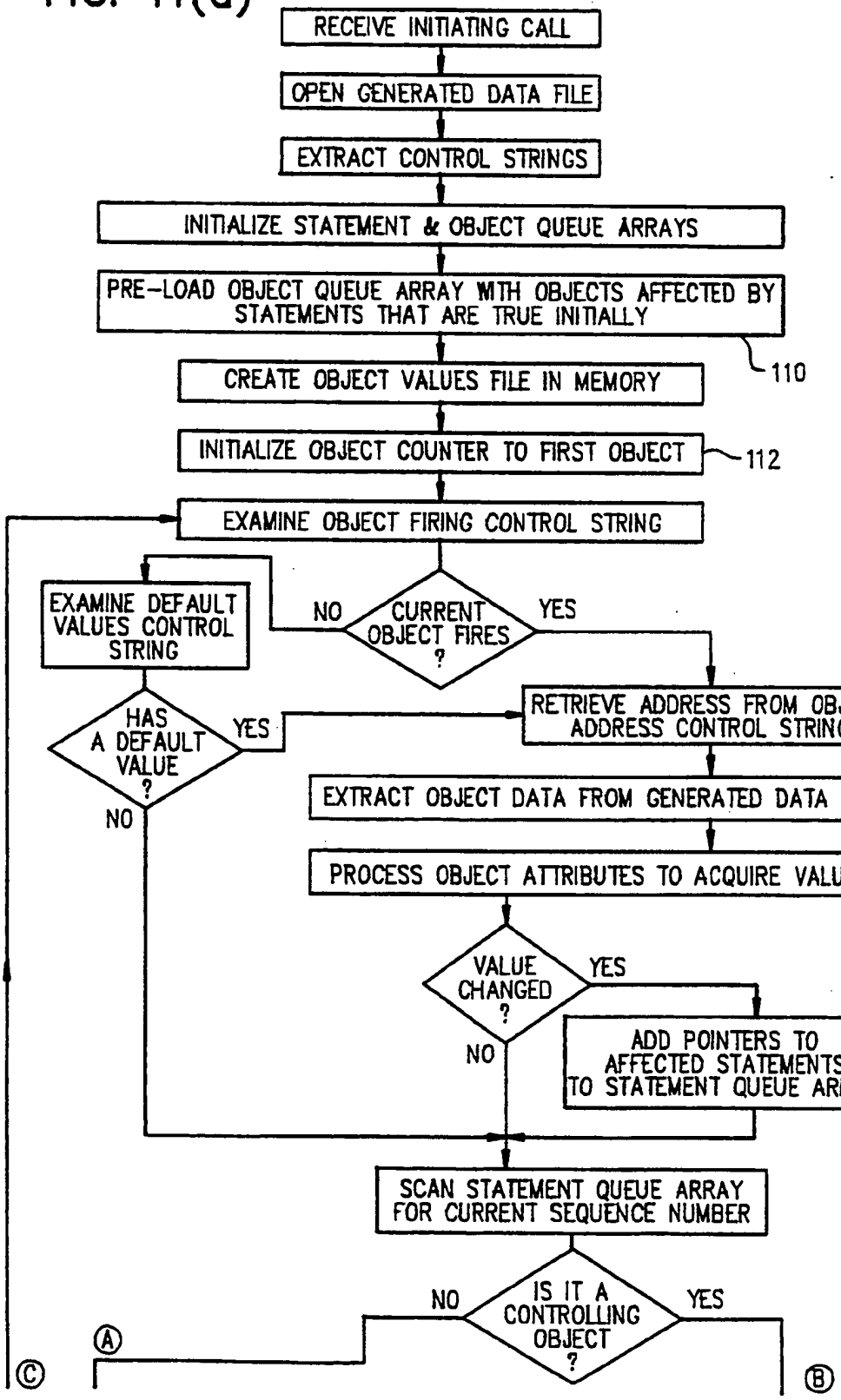

RECEIVE INITIATING CALL
OPEN GENERATED DATA FILE
PROCESS OBJECT FIRING CONTROL STRING
OBJECT FIRES ?
NO
YES
HAS A DEFAULT VALUE ?
NO
YES
ACQUIRE VALUE
UPDATE STATEMENT QUEUE ARRAY
EVALUATE STATEMENTS IN STATEMENT QUEUE ARRAY
UPDATE STATEMENT TRUTH CONTROL STRING
UPDATE OBJECT QUEUE ARRAY
EVALUATE OBJECTS IN OBJECT QUEUE ARRAY
UPDATE OBJECT FIRING CONTROL STRING

FIG. 11(a)


RECEIVE INITIATING CALL
OPEN GENERATED DATA FILE
EXTRACT CONTROL STRINGS
INITIALIZE STATEMENT & OBJECT QUEUE ARRAYS
PRE-LOAD OBJECT QUEUE ARRAY WITH OBJECTS AFFECTED BY STATEMENTS THAT ARE TRUE INITIALLY
110
CREATE OBJECT VALUES FILE IN MEMORY
INITIALIZE OBJECT COUNTER TO FIRST OBJECT
112
EXAMINE OBJECT FIRING CONTROL STRING
EXAMINE DEFAULT VALUES CONTROL STRING
NO
CURRENT OBJECT FIRES ?
YES
HAS A DEFAULT VALUE ?
YES
NO
RETRIEVE ADDRESS FROM OBJECT ADDRESS CONTROL STRING
EXTRACT OBJECT DATA FROM GENERATED DATA FILE
PROCESS OBJECT ATTRIBUTES TO ACQUIRE VALUE
VALUE CHANGED ?
YES
NO
ADD POINTERS TO AFFECTED STATEMENTS TO STATEMENT QUEUE ARRAY
SCAN STATEMENT QUEUE ARRAY FOR CURRENT SEQUENCE NUMBER
IS IT A CONTROLLING OBJECT ?
NO
YES
A
C
B

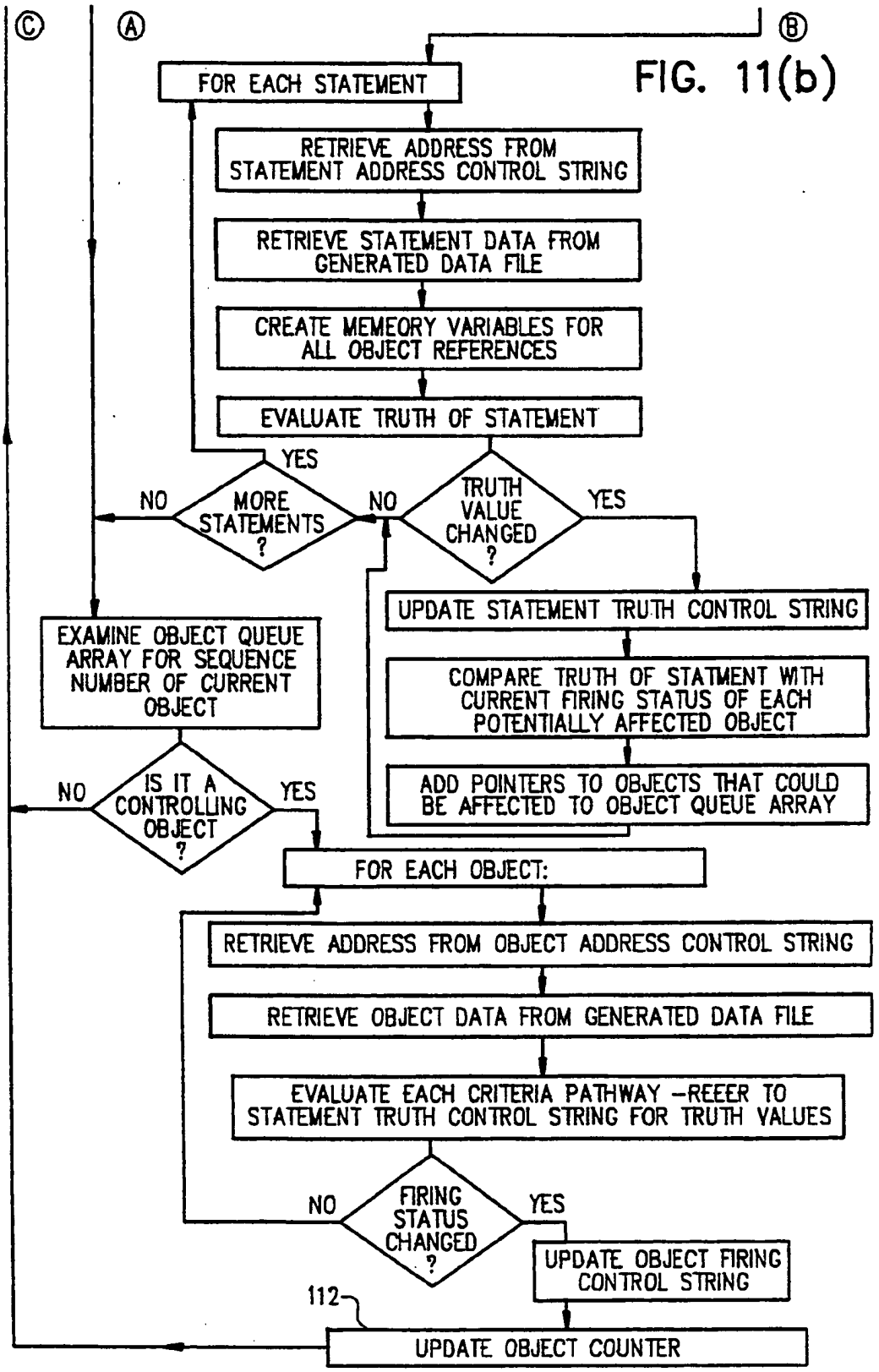
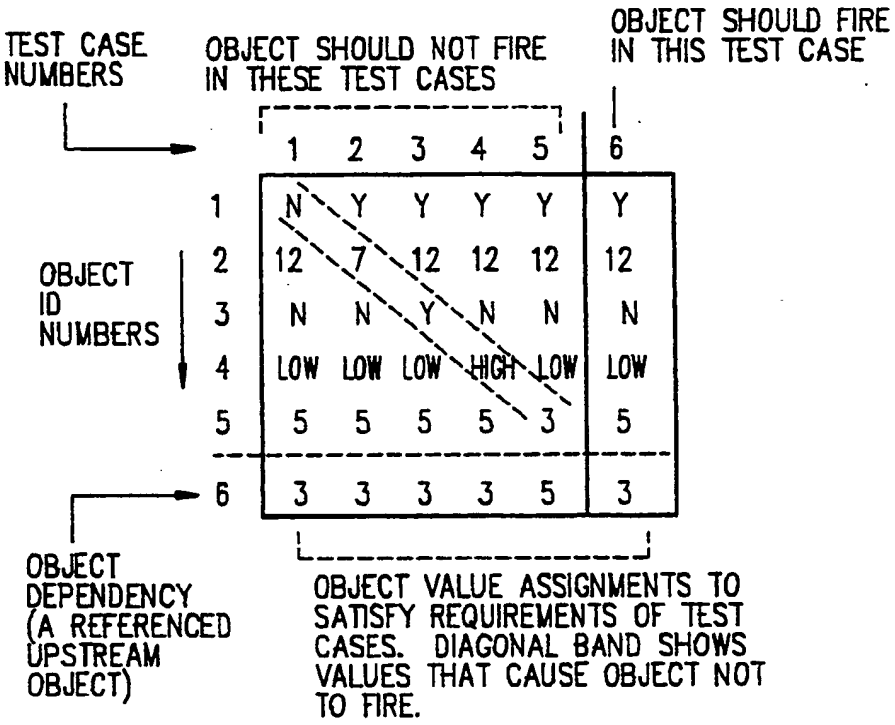

Ⓒ
Ⓐ
Ⓑ
FOR EACH STATEMENT
RETRIEVE ADDRESS FROM STATEMENT ADDRESS CONTROL STRING
RETRIEVE STATEMENT DATA FROM GENERATED DATA FILE
CREATE MEMEORY VARIABLES FOR ALL OBJECT REFERENCES
EVALUATE TRUTH OF STATEMENT
YES
NO
MORE STATEMENTS ?
NO
TRUTH VALUE CHANGED ?
YES
UPDATE STATEMENT TRUTH CONTROL STRING
COMPARE TRUTH OF STATMENT WITH CURRENT FIRING STATUS OF EACH POTENTIALLY AFFECTED OBJECT
ADD POINTERS TO OBJECTS THAT COULD BE AFFECTED TO OBJECT QUEUE ARRAY
EXAMINE OBJECT QUEUE ARRAY FOR SEQUENCE NUMBER OF CURRENT OBJECT
NO
IS IT A CONTROLLING OBJECT ?
YES
FOR EACH OBJECT:
RETRIEVE ADDRESS FROM OBJECT ADDRESS CONTROL STRING
RETRIEVE OBJECT DATA FROM GENERATED DATA FILE
EVALUATE EACH CRITERIA PATHWAY -REEER TO STATEMENT TRUTH CONTROL STRING FOR TRUTH VALUES
NO
FIRING STATUS CHANGED ?
YES
UPDATE OBJECT FIRING CONTROL STRING
112
UPDATE OBJECT COUNTER

FIG. 11(b)

FIG. 12

VALIDATION TABLE



RELEVANCE CRITERIA FOR ABOVE VALIDATION TABLE

LOGICAL OPERATOR	LEFT EXPRESSION	OPERATOR	RIGHT EXPRESSION
	{OBJECT 1}	=	YES
AND	{OBJECT 2}	>	10
AND	{OBJECT 3}	=	NO
AND	{OBJECT 4}	#	HIGH
AND	{OBJECT 5}	>=	{OBJECT6}

A VALIDATION TABLE IS CONTRUCTED FOR EACH CRITERIA PATHWAY. AN "OR" CONDITION CREATES MULTIPLE PATHWAYS, EACH OF WHICH WILL HAVE ITS OWN DEDICATED VALIDATION TABLE. WITHIN A PATHWAY, ALL STATEMENTS ARE LINKED BY "AND".

SYSTEM FOR REVELANCE CRITERIA MANAGEMENT OF ACTIONS AND VALUES IN A RETE NETWORK

BACKGROUND

The invention pertains to methods for programming computers and the software tools for implementing these methods.

Early computer programming methods were limited to procedure based languages in which the actions of the computer were controlled by a long sequence from beginning to end. During execution of the sequence, control might be passed from one program to another which was specified by the first program and then control might be returned to the original program or might continue on to a third. In any event, control followed a sequence established by the programmer.

In order to model the reasoning processes of human experts, expert computer systems were devised which included long lists of facts and long lists of rules stating inferences that might be drawn if and when certain facts exist. When a rule is satisfied, consequences will typically cause the specification or updating of some of the facts. These changed facts will then cause other rules to become satisfied and the choice must again be made of which rule to apply next. In such rule-based systems, there are typically many different rules which are satisfied by any given condition of the set of facts and the application of one rule ahead of another will frequently cause a difference in the result or the speed at which it is achieved.

For this reason, and also because the numbers of rules are so large, applying every rule which is satisfied in each iteration of the system would be prohibitively slow, considerable research and experimentation has been devoted to developing "control strategies" and "inferencing methods" for determining which of the rules should be applied when. For a given set of facts and a given set of rules, different control strategies will produce different results or will reach the results with different efficiencies.

The creation of rule-based expert systems is a complex task, typically requiring a "knowledge engineer" to work with an expert to translate the expertise into long lists of potentially relevant facts and long lists of rules. Once the system is built, it is difficult to verify that the system will produce correct results in each situation and it is difficult to analyze the exact steps taken by the system to achieve each result.

Another architecture for modeling human expertise in computers uses "frames" or "objects" to represent items or classes of items in the real world, and then allows "children" of those frames or objects to be defined which inherit characteristics of their "parents" but are further differentiated from their "siblings" with additional information. Such a system is quite effective for representing taxonomic knowledge such as the method of organizing animal life into kingdoms, phyla, classes, orders, families, genera, and species.

SUMMARY OF THE INVENTION

The invention is a novel method of programming a computer system and the various software tools and components which implement this method. Although the method was developed to meet the need for improved expert systems, it has been discovered that the method is also useful for programming many other types of computer systems and is not limited to use for expert systems.

A fundamental concept of the invention is that of the "object": a necessary fact, a calculated result, a conclusion, or an identifiable element of the expert analysis that is being modeled. Experts do not use the term "objects", but they work with such objects all the time. When human experts use their expertise to solve a problem in the domain being modeled, they think by manipulating such objects. They ask questions to acquire necessary facts for their analysis; they calculate results; they come to intermediate conclusions, which affect their later reasoning; and they calculate or conclude final answers or recommendations. What makes them experts at solving such problems is that they have a long-standing familiarity with these elements of their analysis and they know the relationships among them. They know what questions to ask under what circumstances, and what consequences flow from which facts.

Although this description of the invention employs the word "objects", the meaning of this word has little in common with the "objects" of prior art expert systems.

The invention is designed to eliminate the "knowledge engineer", who serves in conventional expert systems as an intermediary between the domain expert and the system being built. The expert is typically not a computer programmer, and cannot be expected to know how to model his expertise in software, using conventional languages or expert system shells. The knowledge engineer, in the building of a typical expert system, forms this crucial link between the expertise to be modeled and the computer programming that is usually required in order to model it. The knowledge engineer interviews the expert to elicit the expertise and knowledge, and then applies his or her own computer programming expertise to represent that knowledge in the software. Unfortunately, there is enormous potential for error in this process, and it can be frustrating and expensive for all concerned.

The invention allows an expert to model expertise directly in the software. The object paradigm is intuitive enough that most experts take to it relatively easily, and can design their own systems first-hand. To create a knowledge base, one simply creates objects and then specifies the relationships among them in order to perform the analysis. The analysis happens in a specific sequence specified by the expert. This exactly mirrors one of the ways experts solve problems in the real world: they start by acquiring relevant facts, and then "reason forward" in some way from those facts to arrive at the consequences that flow from them. Each fact, consequence and all other factors which are useful in the course of the analysis become objects in the knowledge base.

During execution of a consultation, as facts are acquired from the user, some objects will turn out to be important, while others will be found to be irrelevant to the analysis and therefore will be ignored. In this way the system responds flexibly to different inputs and thereby effectively replicates in software the expertise of the designing expert.

Distinguishing features of the knowledge representation method are that:

(1) Action in the system takes the form of evaluating and firing objects, rather than firing rules as in prior art systems. Objects and their appropriate behavior, rather

than rules and search strategies for applying them, become the central focus of system design.

(2) When an object fires, among other possible actions, it can set a value for itself.

(3) Knowledge is represented by objects and a description of what factors can affect the relevance and appropriate actions of objects. For most of the objects, the relevance criteria include references to values set for themselves by previously fired objects.

(4) The description of all such factors that could affect an object's relevance and appropriate actions are collected in only one place in the form of a set of evaluatable statements associated with the object, the object's behavior criteria.

A distinguishing label for this method is "relevance-actions-value based programming".

The knowledge being represented in the system thus takes the form of an enumeration of the instructions necessary to make a binary decision: whether to take action with respect to an object. The significance of that decision for the analysis being modeled lies in the inherent meaning of the object itself, as intended by the system designer, and in the inter-object dependency relationships that are created by these relevance criteria instructions.

There are important distinctions between the relevance criteria of the present invention and rules of the prior art. Prior art rules are organized in a long list, sometimes with groupings for ease of management, separately from the database of "facts" or "objects". The central question is, given the set of facts, which of many rules that could be applied should be applied? Different search strategies for choosing the rule to be applied can produce different results. New rules can be added at any time without changing existing rules.

In contrast, in the present invention, there is no search for applicable rules. Each object is considered in its turn and its relevance criteria are evaluated. Although the relevance criteria for a particular object can be modified, rules cannot be added to the system.

To translate a rule-based system of the prior art into the relevance-actions-value based system of the present invention, a description of relevant objects must be developed, which will not match one for one with the set of facts or objects in the rule based system, and then each of the rules must be examined to determine whether a portion of the rule should be reflected with an appropriate expression in the behavior criteria for one or more of the objects. With such a translation, some possible results of the original system would not be achievable.

Prior art systems devote processing to navigating a decision tree and reducing the system search space in order to make the system behave more the way we perceive human experts to work, and to achieve greater processing efficiency at run time. But the processing necessary to search rule conditions, navigate the tree, and reduce the search space, is work that the system must perform, and therefore consumes time and system resources. In this invention, a tradeoff is made by committing to examine all objects in the knowledge base in return for avoiding the navigational processing of conventional systems. Yet the same rich network of logic, knowledge, heuristics, and analytical dependencies are represented in the behavior criteria which determine the firing of objects. Performance considerations then focus on ensuring that objects are processed efficiently, particularly those objects that do not fire and are therefore bypassed.

The invention achieves numerous advantages over prior art expert system programming methods. Avoiding the problem of search strategies allows the system to run faster. The use of objects which have behavior criteria and which can be given a value for reference by other objects allows expert systems to be more easily programmed and debugged. The sequence the program will follow and the results that it will achieve are more predictable. The invention allows a consultation session to be easily restarted at any point. As discussed in the detailed description below, the invention allows validation tables to be constructed which allow for easy proof that the system has been correctly constructed.

Although the programming method was invented to meet the need for improved methods for building expert systems, it can be used for programming many other types of computer applications as well. It is particularly well suited to problems which involve multiple decision tree branches with interdependencies and where ease of building, debugging, and validating the system are relatively important compared to the processing speed of the resulting system.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a sequence of objects containing the essential elements of objects in the invention.

FIG. 2 shows a simple form of object processing program which may be used with a data set comprised of objects according to this invention.

FIG. 3 shows the various computer programs, along with their inputs and outputs, which make up this invention.

FIG. 4 shows the various computer files which the object processing program of the invention uses to perform a consultation.

FIG. 5 shows the file structure of the knowledge base.

FIG. 6 shows the file structure of the generated data file.

FIG. 7 shows the data structure of the control strings.

FIG. 8 shows the data structure of the queue arrays.

FIG. 9 shows the data structure of the object values file.

FIG. 10 shows an overview flow chart of the object processing program in the preferred embodiment.

FIGS. 11*a* and 11*b* show a detailed flow chart of the object processing program in the preferred embodiment.

FIG. 12 shows a sample validation table as presented by the validation table generator component of the invention.

DETAILED DRSCRIPTION OF PREFERRED EMBODIMENT

Definition of terms

Application—An application is a set of one or more related expert-system modules that collectively form a conceptual whole and are designed to work together to represent the entire problem domain modeled by the expert system. A given application will have as its centerpiece a database file (the Object Data File 40) containing information about all objects used in the application, and information about each of the modules as well.

Module—A module is a subset of the defined objects of the application that forms a stand-alone expert sys-

tem in its own right. Each module handles one conceptually distinct aspect or area of the expertise needed to be proficient in the overall application domain. In the preferred embodiment, modules are able to call other modules as subroutines in the course of their own processing.

Knowledge base **10**—A knowledge base is the collection of data files in the invention that contains all of the data about the application. These files consist of the Object Data File **40**, the Statement Data File **42**, and the Expression Data File **44**, together with their associated index files.

Object—Objects are the basic building blocks of a knowledge base. Each object represents a fact, question, conclusion, calculation, recommendation, or other material which can, under appropriate circumstances, be asserted as applicable in the reasoning process being modelled. If a piece of data needs to be captured during the course of a consultation, it will be represented by an object.

Attribute—Properties of objects are called attributes. According to the invention, the attributes of every object must include "relevance criteria" **1** and a specification of actions **2** to be taken if, upon evaluation, the relevance criteria are satisfied, one of which may be the setting of a value **3** for the object. In the preferred embodiment, every object also has a descriptive name, which is developer-supplied, and a unique identifying number (the "identifier"), which is system-supplied, as well as an object type designation and an attribute indicating that the object is a member of at least one module. Attributes for a particular object may vary from module to module, among the modules of which it is a member. For example, in one module an object may be an explicit question to be asked of the user, while in another module of the same application the same object might be an internal conclusion which concludes a value 3 based on the values 3 of other objects in the module.

Modules also have attributes, as does the application as a whole. These application-level attributes are stored in the Object Data File in application records, which have an object type of "A".

Object Value **3**—Each object can be given a value which can then be referenced and used by other objects. Object values are stored in a file (see FIG. **9**) created during a consultation.

Consultation **15**—The process in which a user interacts with the developed expert system application is called a consultation. During a consultation, the system processes the objects entered into the knowledge base. Various objects ask for information from the user where necessary, and other objects draw conclusions based on the entered information. After all necessary data has been collected, the consultation typically continues by processing more objects and asserting conclusions or recommendations that are found to be applicable, based on the data collected.

Developer & User—The developer is the person (or group of persons) who creates the knowledge base in order to model the desired expertise. Usually this is the domain expert, the person who has the expertise being modeled. The resulting expert system application is distributed to multiple non-expert users, who benefit from the expertise by using the system.

Generated Data File **14**—Having developed the knowledge base by creating all necessary objects and specifying all of their attributes, the developer generates the Generated Data File from the data in the knowledge base with the Generator Program **8**. The Generated Data File is distributed to users together with the Object Processing Program **9**, which uses it to run the analysis. The knowledge base itself remains with the developer and is not distributed. Modifications to the system are accomplished by altering the knowledge base, generating an updated Generated Data File, and distributing that new file to users.

Composition of an Application

The centerpiece of an application is the collection of files known as the knowledge base **10**. The Generated Data File **14** is generated from these files, and the Object Processing Program **9** uses the Generated Data File in order to run the resulting system.

In the preferred embodiment, the invention is implemented in FoxPro, a microcomputer relational database management system. FoxPro is a variant of the "X-base" family of languages which originated with dbase. The fact that X-base is the de facto industry standard database language for micros means that the invention allows significant connectivity with other systems. This is enhanced by the invention's ability to call custom subroutines from within a consultation, to exchange data with standard external files, and to be called as a module from external programs, such as menu systems.

A consultation can be initiated from other programs by passing parameters which indicate the application number and module number for the consultation. For example, an overall menuing program might have a menu choice for each module in an application, other choices for modules of a different application, and still other choices for subroutines that are not consultations.

Several other types of files are usually needed in order to constitute the overall application in its final form. Typically there are other data files that are created by the developer to hold data for a historical record of prior analyses. These can be crucial to the correct performance of the system, since the invention allows values to be imported from such external files and assigned to objects in the course of processing a consultation. The invention also can take a consultation **15** (partial or complete) in its entirety, including answers supplied by the user, and save it in compressed form to a memo field in a particular record in such an external file. This allows consultations to be replayed at a later date.

In addition to external data tables, a finished application will usually include a file of custom subroutines which may be called by objects during the course of processing a consultation. The invention can open a channel to such a file in order to allow rapid access to such subroutines by opening it as a FoxPro procedure file at the start of the consultation **15**.

Contents of the Knowledge Base

The knowledge base **10** consists of the objects within an application and their attribute data. The knowledge base holds all information about the objects, their characteristics, their interrelationships, the external files they interact with, and the customized subroutines that they may execute.

The knowledge base is contained primarily in the Object Data File **40**. Each record in this file holds an object. Each object must have a name, one that should be intuitive to the developer. It will also have a system-generated, unique identification number (the "identi-

fier") used by the invention. It will also have various attributes that distinguish it from other objects. One fundamentally important attribute of each object is its "relevance criteria" 1: the circumstances under which the object is considered to be relevant to the analysis. (An object that is determined to be relevant is said to "fire".) Other attributes include actions that an object may take, and action criteria which describe the conditions under which the actions should be executed.

Other files support the Object Data File 40 to form collectively the entire knowledge base. These are:

the Statement Data File 42, which holds unique statements used in relevance and action criteria, and

the Expression Data File 44, which contains all unique values that objects may acquire, as specified in the Object Data File, and unique left-side and right-side expressions used in criteria statements.

As shown in FIG. 5, the Object Data File 40 has the following structure:

Field	Type	Length
ID	Character	4
NAME	Character	10
ATX	Character	20
ATT	Memo	10 (a pointer to variable-length text in an associated file)

The ID field stores unique numeric object identifiers. The Statement Data File and the Expression Data File contain similar identifiers. New identifiers are assigned as needed by finding the largest identifier in current use and incrementing its value by one. To facilitate the use of such identifiers as pointers, all identifiers are stored as character strings composed of numeric digits, e.g. "0123" or "6397". The preferred embodiment uses 4-character strings, yielding a system capacity of up to 10,000 identifiers in each file ("0000" through "9999"), but identifier strings of any length may be used.

The NAME field stores each object's name. Every object in the knowledge base must have a unique name, assigned by the developer upon its creation. A name should be descriptive of the meaning of the object, or the role it plays in the analysis being modeled, but is limited to FoxPro's maximum length of 10 letters for memory variables, so that such variables may be created using these names if desired. An implementation of the invention using a language other than FoxPro would likely have a different constraint on the allowed length of a memory variable name.

The ATX field stores temporary attribute codes extracted from the ATT field. These codes are used for indexing, to facilitate the handling of objects and the editing of their attributes. When a module is opened so that objects in the module may be edited, certain attribute codes are extracted from the ATT data for the module and stored in the ATX field, updating the index files. For example, one of the attributes stored in ATX is the object type code: this allows all objects of a given type to be grouped together and viewed as a single collection of objects.

The ATT memo field stores all attribute data (other than ID and NAME), in the form of character strings. Where an object is a member of more than one module, module delimiters (described below) separate one module's data from the next module's data. Within an attribute data string, attribute delimiters separate one attribute's data from the next.

System Organization

The developer of an expert system application using this invention creates objects and organizes them into a desired sequence. The sequence usually flows from the approach used by the expert when solving a problem, and the resulting sequence will reflect the progression of the expert's thinking when analyzing the problem. This close mapping between the flow of logic in the system and the natural approach taken by the expert is of great practical benefit in system design and testing, making it possible for the expert to model his or her own expertise directly in the system.

Object sequence is also important from the user's point of view. Questions asked of the user, and messages from the system to the user, will vary from consultation to consultation, as different objects are determined to be relevant under different fact patterns. It is beneficial to present the questions and messages to users in as consistent an order as possible, so that over time the user will find the behavior of the system familiar and internally consistent. This can be a factor in the confidence users place in the system, and in its ultimate acceptance.

When the Generated Data File 14 is created from the knowledge base 10 by the Generator Program 8, the invention will resequence objects if necessary in order to ensure that object dependency relationships are respected, but such resequencing is kept to a minimum in order to preserve the developer-defined object sequence to the extent possible.

Knowledge Base Delimiters

Several delimiter conventions are used in the invention to separate attribute data within an object and to identify module information.

Module delimiter—this identifies data for a particular module. It takes the form of a tilde ("≈"), followed by a twodigit module number. For example, data for module 7 would appear in this way ≈07<data>.

Attribute delimiter—this marks the beginning of an attribute data string. It consists of ASCII character 4 (" "), followed by a two-digit numeric code identifying the attribute. For example, the text of an on screen object is assigned attribute number 5. It's attribute delimiter is: 05<text data>.

Internal data delimiters—The invention uses several characters as delimiters to separate data elements within attribute data strings. These include the vertical bar ("|"), and backslash ("\").

Criteria delimiter—where criteria are coupled with a particular data element (for example, when one of several possible values for an internal conclusion has associated action criteria), ASCII character 254 ("■") separates the data element from the criteria string.

Objects and Their Attributes

An object represents any individual element of the analysis being modeled in the expert system.

An object could be, but is not limited to, any of the following:

(1) any useful concept, idea, abstraction, or judgement,

(2) any fact that the human expert in the problem domain might need to know in order to solve the problem,

(3) a representation of a real-world entity, such as a person, a machine, or an organization,

(4) any question to be asked and answered by the user,

(5) any conclusion to be drawn if specified facts are present, including intermediate conclusions and the results of calculations, or

(6) any answer, recommendation, message, or other result that might be produced as useful output in the course of the analysis.

Each object has a collection of assigned attributes which, in a typical system might include:

(1) ID: A unique identifying number.

(2) NAME: A unique name, which is an intuitive descriptor defined by the developer to connote the object's meaning.

(3) TRANSLATION: A meaningful, short, but usually multi-word description of the object, which is displayed during execution in various contexts as a replacement for the more cryptic object name.

(4) OBJECT TYPE: An object's type is a mandatory attribute of every object which determines the standard actions that are performed in order to process it appropriately when it fires. The Object Processing Program will recognize the object type attribute for a firing object and will call the necessary subroutines in order to process it appropriately. Objects that are common to multiple modules in an application can be of different types in the different modules. Within a module, an object may be assigned multiple types (for example, a message object may also be designated an output object, so that the message text is output to a file after processing).

(5) SEQUENCE: The initially preferred sequence for evaluation of each of the objects within the consultation is specified by the developer. Developer sequences are respected where possible, to allow maximum control over the order in which onscreen objects are presented to the user. However, if an inconsistency is discovered between a developer-defined sequence and an object dependency relationship when the Generated Data File is being generated, objects will be resequenced by the Generator Program 8 as required in order to satisfy the object dependency relationships involved, i.e., ensuring that all objects on which a given object depends precede it in the resulting sequence.

(6) TEXT: The text of a question or recommendation or message that is concluded to be applicable if and when the object fires.

(7) POSSIBLE VALUES: Possible menu choices that might be presented to the user when a question is asked on screen, or alternative values that might be concluded by an object, including constants, literal strings, mathematical formulas to be evaluated, default values, or references to other object values.

(8) DATA TYPE: The data type of the object's expected value to be concluded.

(9) LINKS: Instructions for interactions between the object and other entities, such as importing a value from an external data file, exporting data to such a file, or calling a subroutine.

(10) RELEVANCE CRITERIA: A set of statements to be examined when the object is evaluated. If, upon evaluation, the relevance criteria are satisfied, the object will "fire" at the appropriate time in the sequence by taking the actions of performing the appropriate action for the object type, setting a value for the object, and performing any specified links.

(11) ACTION CRITERIA: A set of statements to be examined after an object has fired in order to determine if possible actions are to be performed. If, upon evaluation, the action criteria are satisfied, then the action will be performed. Not all actions have action criteria; where they do not, such actions are always performed.

Types of Objects

There are no hierarchies or other classifications of objects for purposes of knowledge representation. In particular, realworld knowledge is not explicitly represented through object classes, as is the case in prior art "frame-based" or "object-based" expert system development tools. The objects in the knowledge base are separate, atomic units on a peer level with one another, and do not inherit attributes from one another.

Every object in the knowledge base will have as an attribute at least one assigned object type designation. The possible types of objects are:

system object: go-to
system object: reset
system object: inter-module call
demon object (calls a subroutine if it fires)
input object: screen—user input
input object: screen—select one from predefined menu
input object: screen—multi-object list
input object: screen—message object
input object: conclusion—import
input object: conclusion—conclude a value
output object—list ID to file
output object—list text attribute to file
output object—export object value to file

An object can be designated as being of more than one type. For example, a message object is designated as a screen object type and will appear on screen as an input object during the consultation, and it may also be designated as an output object, so that its text is used for some purpose at the end of the consultation.

Input Objects

The class of input objects is divided into two subclasses, screen objects and internal conclusion objects. Screen objects ask questions to elicit information from a user of the system, and may require direct data entry or a selection from a menu of alternatives. They may also post messages on screen, to advise the user and make recommendations. Internal conclusion objects acquire values according to predefined settings for the Possible Values attribute, which may refer to the values of other objects, or by importing values from external sources, such as a database file.

Screen objects

These are objects which acquire a value supplied by the user. Text appears on screen, and the user supplies an answer. The text of a screen object can contain embedded values, including live data from the current consultation. There are four types of Screen objects: user-input, select-one, messages, and multi-object lists.

(1) User-input screen objects

These objects require the user to type in an answer. For example, if the consultation requires a figure for the dollar value of a contract, the knowledge base should have an object that asks the user something like "What is the value of the contract? " in order to acquire this value. The text of the question to be displayed on screen is stored as the text attribute of the object. Another

stored attribute specifies the number of spaces or columns on screen that the system should provide for the user to type in the answer to the question.

(2) Select-one screen objects

These are objects which ask the user to select one answer from a pop up menu of possible choices. The text of the question is displayed and the menu of choices appears, allowing the user to make a selection. Unique menu alternatives are stored in the Expression Data File 44, and pointers to such values are stored in the Object Data File as the Possible Values attribute of the object. In this way, menu alternatives are only stored once, and may be reused by other objects by using pointers to the alternatives.

Menu choices can be suppressed from appearing on a menu by assigning action criteria to the menu choice. At runtime, these criteria will be evaluated when building the menu, and if the criteria are not satisfied, the choice will not appear. This allows menus to behave in a context-sensitive manner, responding to the particular facts of a consultation, so that irrelevant choices are not offered to the user. Where no action criteria are specified for a menu choice, the choice always appears on the resulting menu.

(3) Message screen objects

These objects are displayed on screen, but they are not questions and do not accept any response from the user other than pressing a key to continue. Message objects are used to display information to the user or to post notices that call the user's attention to some fact or result. The information displayed could be advice, recommendations for actions, or the interim results of a particular analysis that has been performed by the system. Frequently, a message object will have the current values of one or more other objects embedded in its text.

Message objects are quite useful for communicating the progress and the results of the system's analysis as it proceeds during the consultation. Additionally, they are useful aids for developers when debugging the knowledge base. Often, message objects are also designated as output objects, so that their text is asserted as applicable in some fashion at the end of the consultation, e.g. in a report containing consultation results. If they are so designated, their formatted text (including any embedded values) is preserved to become the object's text as an output object.

Message objects do not acquire values per se, but are assigned a value of "<System Message>" by the system. This allows them to be viewed and recognized as messages when the user is reviewing the values of objects that have been processed, and allows their values to be referenced by other objects if desired.

(4) Multi-object lists

A multi-object list presents the user with a list of items, each of which represents an object within the system, and allows a "mark all that apply" approach, instead of requiring a single choice from a menu. In a multi-object list, each item presented on the list of choices is an object in its own right. Each object has its own relevance criteria which determines its applicability. If these criteria are satisfied, the object will appear on the resulting list; if not, the object will not appear.

Objects to be included in a multi-object list have as an attribute the number of the particular multi-object list to which they belong. During the consultation, such objects are processed in the order encountered to determine their applicability, and a list of the successful objects is maintained in memory. The final object in a multi-object list has an attribute which indicates that it is the last object. After processing this terminating object, the resulting list of applicable objects is presented to the user.

Internal conclusion objects

Internal conclusion objects acquire values just as screen objects do, but they do it transparently, without the involvement of the user, based on previously entered information. Conclusions can have one or more possible values to conclude or calculations to perform, as specified by the Possible Values attribute, and may use the values of other objects in the application. Conclusion objects may adopt the value of another object, perform a calculation using other objects' values, accept a literal string as a value (for example, "YES" or "Sell the stock"), call a custom subroutine to assign a value, or import a value from an external file.

An internal conclusion object may contain an expression to be evaluated, and the result of such expression becomes the value concluded by the internal conclusion object. Unique possible values or expressions are stored as records in the Expression Data File 44, and pointers to these records are stored as the Possible Values attribute of the internal conclusion object. Potential values to be concluded may have specified action criteria, which are evaluated in the order that the values are defined. The first successful value is taken as the value of the object, and the remaining values or expressions in the list of possibilities are not considered. This allows possible values to be prioritized. Where a potential value has no action criteria specified for it, the value is always concluded.

Internal conclusion objects may alternatively import data from another source, instead of using predefined values. Instructions for performing the import are assigned as an attribute of the object. These instructions include the name of the data file to be used, an associated index file name, an expression to be evaluated to serve as an index key for locating the desired record in the external file, and an expression to be evaluated to acquire the value to be imported (usually comprised of one or more fields in the external data file). Imports may have action criteria defined for them. At the time the import is to be performed, such action criteria are evaluated and the import is performed only if such criteria are satisfied.

Output objects

Output objects are a powerful way for the system to report answers or conclusions drawn during the consultation. The knowledge base can be configured to evaluate the applicability of various possible alternative statements or recommendations, based on entered facts, and assert them in some fashion at the end of the consultation.

System objects

System objects achieve basic system operations for program control in special situations. If the consultation should be terminated and started over, a "reset" system object can cause the values of all objects to be released, and the system to be started again in its initial state. System objects can cause other modules to be run, with control then returning to the calling consultation, and can cause the consultation to jump to a specified object under given conditions.

Application Level Attributes

Every Object Data File 40 contains an application record in addition to object records. The application record contains control information and attributes about the application as a whole, and attributes for each module, as distinguished from object level attributes. For example, the name given by the developer to Module 6 is a module level attribute. It is the text string that the system will extract and place in the upper left corner of the screen when executing the consultation for Module 6.

The application record is distinguished from regular objects with an "A" in the OBJECT TYPE attribute. The NAME attribute for the application record contains a numeral (for example, "1"), which is the application number.

The application record stores:

- the title of the application as a whole,
- a version number for the entire knowledge base,
- the names of application-specific external files which should be opened at the outset of a consultation and left open to facilitate the exchange of data between the system and such external files,
- the name of a default global help file for the application, and
- module-specific attributes

Module-specific attributes in the application record are:

- the module number,
- the title of the individual module,
- a version number for the module,
- the date and time of the most recent generation of the module's Generated Data File 14,
- the name of a procedure file to be opened, to allow access to custom subroutines to be used by the module,
- the name of a module-specific help file,
- instructions used when looking for restorable consultations, and when saving consultations, (These instructions include the name of an external file where consultations are stored, expressions used in order to construct a menu from which to select a particular consultation to be restored, and an expression used in order to construct an index key to be used at the end of the consultation to locate the appropriate record for consultation storage.) and
- instructions for constructing and refreshing a status display which is shown in the upper right corner of the screen during a consultation.

Relevance Criteria

Knowledge is represented in the system by the meanings of the objects that are created, and additionally through the use of a fundamental attribute of every object: its defined collection of "relevance criteria", which consists of one or more declarative statements. Statements take the form of a left-side expression of any complexity, a conventional operator, and a right-side expression of any complexity, and can be evaluated for their truth value. These relevance criteria statements may refer to the values of other objects in the system, thereby creating inter-object dependency relationships.

Multiple statements in the relevance criteria set for an object are linked by the Boolean operators "AND" and "OR". Statements may be grouped algebraically in an arbitrarily complex fashion, with nested parenthetical groupings allowed, and may be assigned a required certainty factor which must be satisfied in order for the statement to evaluate to "true" (see "Certainty Factors" below). The resulting set of statements in the relevance criteria collectively form a complete description of the circumstances under which an object should be considered relevant and applicable to the analysis which the expert system is designed to perform.

For example, if X is an object, a typical statement in its set of relevance criteria might be: "A>B". This creates a dependency relationship between X and the objects A and B, for X's applicability in the system will be influenced by the values of A and B. The statement can be evaluated for truth, using the values of A and B and applying the ">" operator to compare them.

The groupings of statements, possibly nested, and alternatives among them (signified by a linking "OR" Boolean operator) are expanded by the Generator Program 8 during the creation of the Generated Data File 14 in order to form distinct, alternative, enabling "pathways" for the success of the criteria. All statements within such a pathway are connected by "AND", so that all statements must evaluate to "true" in order for the pathway's requirements to be satisfied. Any one such pathway for an object—when all of its statements evaluate to "true"—is sufficient to designate the object as relevant and applicable. If, upon evaluation, at least one of an object's enabling pathways does succeed and the object is therefore considered to be applicable, it is said to "fire". When an object fires, the consequence is simply that the Object Processing Program 9 ought to take appropriate action with it. What the object does when it fires is determined by its other defined attributes.

Certain objects in an application may always fire. For example, some initial data must always be collected in order to begin the analysis. Such objects are treated as special cases and are not assigned any relevance criteria. The absence of relevance criteria in this context is an indication to the Object Processing Program that the object should always fire.

The concept of relevance criteria is extended in the system beyond its fundamental application of describing the circumstances under which an object should fire, to describing the circumstances under which particular attributes of objects should operate in alternative ways. Such criteria sets are called "action criteria". For example, in the case of an internal conclusion object having some number of alternative values, each value will typically have, as an attribute of the object, a cluster of action criteria that specify the requirements for that value to be used. The action criteria statements for a given value are expanded if necessary to identify alternative, stand-alone pathways, any one of which is sufficient to cause the value to be assigned (concluded), and in each case the action criteria statements form a set of instructions for making the binary decision of whether to utilize the value or ignore it. By convention, the first successful value in a list of alternative values will be employed, and the remaining members of the list will not be evaluated. This allows the developer to prioritize the conclusion alternatives by organizing them in a preferred sequence.

In the case of a screen object with alternative menu choices, action criteria (if any) associated with a particular menu choice must be satisfied in order for that option to appear on the resulting menu, else it will be omitted. By convention, in this context all members of the list of values are processed, and values having no

associated action criteria always appear in the resulting menu.

Internal conclusion objects may also acquire values by importing data from external sources, such as a database file. Action criteria may be specified to describe the circumstances under which such an import is to be attempted.

Other examples of extended uses of the criteria concept include describing the circumstances under which:

(1) a specified procedure or subroutine should be executed

(2) one of several alternative phrasings of text should be employed

(3) a value from one or more other objects should be dynamically embedded in the text of a given object to form a customized message.

Processing all objects and firing them on the basis of an evaluation of their relevance criteria produces a subset of the total objects in the knowledge base, a list of objects which are all known to be applicable (they successfully fired). This list can be thought of as a collection of objects in the knowledge base which is customized to fit the current fact pattern. The input objects in this set of successful objects have acquired all relevant information and have reached appropriate conclusions. All output objects in the set specify output actions that have been taken, such as asserting their text as answers and recommendations, and many others.

Output objects are usually blocks of text (of any length), and many expert systems are designed specifically to produce such text. The text could be something like "Check to see that the unit is plugged in.", or it might say something more significant, such as "Fire the missile and start the war." These are essentially just the communication of a result to the user of the system. "User" in this sense might be another system, rather than a human, and so the output object may indeed cause an action, e.g. causing some block of code to be executed, such as a subroutine, or an instruction to return a value to a calling program.

Alternatively, actual work product might be the goal: e.g., an "intelligent document assembly" program might produce an actual document, such as an insurance policy customized to the facts, risks etc. involved in a particular person's application for insurance. It might perform the analysis according to decision rules and company policies which are represented through objects' relevance criteria, and build the document using standard company language, embedding data values at specified places in the text. In such a case, what may happen is that the system should simply use the text of the successful objects to create the resulting file directly. Alternatively, the objects may just consist of pointers to files or blocks of text in other systems, and this system may just list the identifiers of the successful objects to a file. A follow-on program might pick up the resulting file and look to other resources to build the end product.

All objects on the list are considered, and based on their relevance criteria, some objects are listed as applicable under the known facts while others are rejected as inapplicable.

Often a knowledge base will include several objects to represent alternative outcomes for a given concept or result. For example, in an application for medical diagnosis one object's text might say "The available evidence indicates that this patient has Parkinson's disease." The following object's text might say "The available evidence indicates that this patient probably does not have Parkinson's disease." Clearly, these two objects should not both fire, and would have mutually exclusive relevance criteria. In the context of a particular set of facts, if one of these objects fires and is therefore asserted as applicable, its counterpart should not fire. Depending on the relevance criteria specified, it might also turn out that neither object will fire, perhaps because a previous object has ruled out the possibility of Parkinson's disease altogether.

The "Table View" of Criteria

Criteria statements are specified by the developer of a knowledge base using what is called a "table view". In this representation, each statement occupies a row in a matrix. There are columns for the Boolean operator, left-side parentheses for algebraic grouping, a left-side expression, a connecting operator, a right-side expression, right-side parentheses, and a certainty factor threshold for the statement. Left-side and right-side expressions in a statement may consist of any syntactically valid expression in the language being utilized, and often contain references to other objects. An object reference is specified by the developer by using the object's name, and enclosing the name in curly brackets ("{}").

All criteria statements (except the first) must be linked to preceding statements by a logical operator, either "AND" or "OR". Parentheses are used for algebraic grouping of statements, and may be nested up to five levels deep. Parentheses are also allowed within the expressions used in a statement, to permit normal algebraic grouping within such expressions. Certainty factor thresholds for individual statements are optional. Where omitted, the statement is treated as having a 100% certainty factor (see "Certainty Factors" below).

There are eight allowed connecting operators in statements, and each is assigned a number, as follows:

Number	Operator	Meaning
1	=	equals
2	#	does not equal
3	>	is greater than
4	<	is less than
5	>=	is greater than or equal to
6	<=	is less than or equal to
7	$	is contained within string
8	!$	is not contained within string

The last two operators work with comparisons of character strings. The "$" operator evaluates whether the left-side expression, comprised of a character string, is contained within the right-side expression, which is also a character string. Thus, the statement

"bcd" $ "abcde"

evaluates to "true", and the expression

"xyz" !$ "abcde"

also evaluates to "true".

Here is a generic example of a complex set of relevance criteria for an object:

	Logical Operator	Left Paren	Left Expression	Connecting Operator	Right Express.	Right Paren
a)			{OBJECT1}	=	{OBJECT2}	

-continued

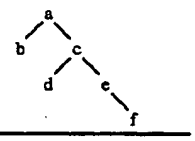
	Logical Operator	Left Paren	Left Expression	Connecting Operator	Right Express.	Right Paren
b)	AND	(	{OBJECT3}+{OBJECT4}	>=	50000	
c)	OR		{OBJECT5}	#	({OBJECT6}+{OBJECT7})/2	
d)	AND	(	{OBJECT8}/2	<	100000	
e)	OR		{OBJECT9}	>	0	
f)	AND		{OBJECT10}	=	YES	))

The letters in the left column are labels designating the different criteria statements, to facilitate discussion.

When the Generated Data File **14** is generated by Generator Program **8**, such criteria statement sets are evaluated and expanded to form standalone "pathways" for the independent ways that the criteria may be satisfied. In this expansion process, individual statements are treated as indivisible units, and are manipulated to create pathways by distributing common statements across parentheses groupings. The resulting pathways no longer contain "OR" terms, or left and right parentheses to group statements (although within a statement, any parentheses within an expression will remain unchanged).

In the example above, there are three possible ways that this object's criteria could succeed, causing the object to fire. Using line labels to represent the statements, the resulting pathways are:

Pathway	Successful statements
1	a,b
2	a,c,d
3	a,c,e,f



Firing Bias

The collection of relevance criteria statements for an object has an implicit result that incorporates default behavior for the binary choice of whether the object should fire. In the preferred implementation, this result is that if the criteria succeed, then the object should fire. As presented, the default is always not to fire the object, and the relevance criteria define the circumstances under which this default behavior should be overridden and the object caused to fire. This is referred to as the "firing bias" of the object: the system is biased against firing objects, and specified criteria must be satisfied in order to overcome this bias and affirmatively assert the applicability of the object.

Extensions of the criteria concept within the system—for example, use of action criteria to control which value should be concluded or whether an import should be attempted—use the same "affirmative" approach: action criteria specify the conditions under which something should happen, with the default being that the action should not be performed.

An alternative embodiment of the invention incorporates the opposite default behavior. In such a system, the firing bias is reversed, so that an object will always fire unless its relevance criteria statements are satisfied, in which case the object will not fire. Thus the relevance criteria become a description of when not to fire the object, rather than a description of when to fire it. In this embodiment of the invention, an application-level attribute specifies which firing bias to use, so that the Object Processing Program **9** behaves appropriately.

There is no inherent reason to prefer one firing bias over the other. In fact, overall system storage requirements are minimized if the two biases are both used and applied on an object by object basis, so that some objects have one firing bias while other objects have the opposite firing bias. In this alternative embodiment, an object-level attribute specifies the object's firing bias. If an object typically will fire in most cases, it may be more economical to represent the knowledge about its behavior by enumerating the circumstances under which it should not fire, i.e. by giving it a bias toward firing. Alternatively, if an object generally should fire only in specified circumstances, it may be easier to identify those circumstances and give it a bias against firing.

However, the mixture of two firing biases introduces significant maintenance problems in the system. Developers of a system find it much easier to deal with a consistent firing bias, and confusion can therefore be avoided by adopting a single firing bias and applying it consistently throughout the system.

Storage of Criteria

When specifying criteria for an object, the developer creates evaluatable statements. As shown in FIG. **5**, unique statements that have been created are stored in a separate table (the Statement Data File **42**), and are assigned a unique identifying number, using the same identifier scheme as was used in the Object Data File **40**. Left-side and right-side expressions used within statements are stored in the Expression Data File **44**, again with a unique identifier number.

Criteria sets are stored in the Object Data File. Statements used in criteria may be reused by different objects, and expressions used in statements may be reused by different statements. Consequently, statements are stored and referenced in the system separately from the objects which use them, and expressions are stored and referenced separately from the statements which use them.

The identifying numbers in each file serve as pointers among the three data files. Criteria sets in the Object Data File contain pointers to the applicable statements in the Statement Data File, using the statements' identifiers. A given expression is stored only once, in the Expression Data File. When it is used within a statement, the statement records the usage by using the expression's identifier as a pointer into the Expression Data File. Because a statement always consists of a left-side expression, an operator, and a right-side expression, an encoded representation of the statement as a 9-digit numeric string is possible:

<expression identifier> <operator number> <expression identifier>

EXAMPLE: "042530379"

In this example, "0425" is the identifier for the left-side expression. The expression itself may be retrieved by locating identifier "0425" in the Expression Data File and retrieving its stored expression, which might be something like "{OBJECT 63}+{OBJECT_09}". The "3" in the fifth position of the string stands for the third connecting operator, ">". Finally, the "0379" is the right-side expression's identifier, which when located in the Expression Data File might yield an expression such as "100,000". The complete statement therefore expands to become:

{OBJECT_63}+{OBJECT_09}>100,000

This is a statement which can be evaluated for truth, using values acquired during processing of the system. Such a statement can become one of an arbitrary number of statements in a given object's set of relevance criteria, and can be reused by other objects in their criteria sets as well. The 9-digit string representing this statement in encoded form is stored in the Statement Data File, and is assigned its own identifier, such as "2947". Because the expressions within a statement are referenced by pointers to the Expression Data File, the reference to the value "100,000" in this example can be reused by many other statements as well.

Objects store criteria in the form of pointers to statements such as the example above, using the statements' identifiers. If an object's firing behavior depended solely on the example statement above, its relevance criteria would consist of the string "2947", i.e. this statement's identifier. Boolean operators which link statements in criteria sets are represented by an underscore ("_") for "AND", and a period (".") for "OR". Parentheses used to group statements in a set of criteria are stored literally. Thus, suppose object X had the following criteria:

		<statement 1, having identifier number: 0123>	
AND	(	<statement 2, having identifier number: 0234>	
OR		<statement 3, having identifier number: 0345>	)

This set of statements would be stored in the Object Data File as the character string:

0123_(0234.0345)

Other means of representing statements and expressions are certainly within the scope of the invention. For example, storage requirements for the knowledge base could be reduced by using a different pointer convention, where pointers are of shorter length.

Creating the Knowledge Base

The preferred implementation provides a computer program (the Knowledge Base Development Program 4) to facilitate the creation of applications, modules, and objects, and the editing of their attributes. A developer who wishes to create a new application selects "Create New Application" from a menu, and after certain information such as the application name and number are entered, the program creates the necessary files. A new module for an existing application is created in a similar fashion: after the developer provides a name and number for the new module, this data is added to the Object Data File's 40 application record.

Objects are always created and edited within the context of a particular module within an application. In order to work with objects, the developer opens the application and module of interest by selecting from a menu. "Opening" an application causes its files to be placed in use, and "opening" a module causes indexing information to be updated for all objects in the application's Object Data File 40. The ATT field of the Object Data File contains all attribute data for each module of which an object is a member. (see FIG. 5.) Those objects which are members of the module being opened have certain attribute data extracted from the module's data segment in the ATT field, and this data is placed in the ATX field for use in indexing. Objects which are not members of the module being opened will have a blank ATX field, for there is no data from the ATT field to extract and insert into the ATX field. This allows records with blank ATX fields to be filtered out, and the resulting set of records which are in use and available for editing are just those objects which are members of the opened module.

When a module has been opened, an "action" menu appears, giving the developer a choice of several alternative actions to perform. Options on this menu that relate to working with specific objects in the knowledge base are "Edit", "Create", and "Delete". Other options on the menu offer various utility and diagnostic services, such as a utility to create a report which lists the contents of the knowledge base for this module. Other possible options are the Validation Table Generator 5, the Natural Language Interpreter 6 and the Explanation and Diagnostic Utility 7. Another option on this action menu is "Create the Generated Data File", which invokes the Generator Program 8 to put information from the finished knowledge base in a file format suitable for processing the module as the intended expert system. Such processing will be performed by the Object Processing Program 9.

To edit the attributes of an object, the developer selects "Edit" from the action menu and is then presented with a menu of existing objects in this module. The developer has options for what objects are presented and for how this menu should be displayed. For example, the developer might direct the menu to display only internal conclusion objects, and to display them in alphabetical order rather than in their defined sequence order.

Selecting an object from this list opens that object for editing, and causes a menu of object attributes to appear. The developer selects the particular attribute to be edited for this object. Depending on the attribute involved, an appropriate interface is provided to facilitate the editing. For example, if the text of a screen object is to be edited, then a window opens showing the text that is currently assigned as the object's text attribute. The developer can make changes to this text and then save the new version.

To create a new object, the developer selects "Create" from the action menu, and must then enter the proposed name and sequence position for the new object. The system then checks to see if the object already exists. Several variations are now possible:

(1) If the object does exist and it is already a member of the module being edited, the creation action is rejected, for duplicate object names are not allowed.

(2) If the object exists as a member of one or more other modules, the effect of the creation action is to include it as a member of the current module as well. The assignment to the object of at least one attribute for the current module is enough to make it a member of the module. In this situation, the developer has the opportunity to copy the attributes of the object that have been assigned for purposes of another module to become its attributes for the current module as well, saving repetitive data entry.

(3) If the proposed object does not yet exist, a new record is added to the Object Data File **40** for it and a new system identifier number is assigned to it.

After an object is added to the current module, the normal editing process for its attributes begins.

Object deletions are handled in a similar manner. The developer chooses "Delete" from the action menu, and then selects from a list of current objects the object to be deleted. If the object is a member of the current module only, its record is removed from the Object Data File. If it is a member of one or more other modules as well as the current module, the developer is asked if the object should be removed from all modules, or only from the current module. If it is to be removed from all modules, the record is deleted from the Object Data File. If it is to be removed from this module only, its attribute data for this module is deleted, but the object's record is retained, and its attribute data for other modules is unaffected.

When an object is added or deleted from a module, the developer-defined positional sequence numbers for all objects in the module are adjusted to account for the change. When an object is deleted, all objects which contain references to it are visited and such references are removed, in order to maintain referential integrity among objects. If the deleted object was a member of other modules as well, and it was deleted from all modules, all objects in those modules must also be visited in order to remove references to it. If it was deleted only for purposes of the current module, this process is limited to just those objects within the current module.

Most attribute data that is entered for an object is stored directly in the object's ATT field in the Object Data File. For example, if a conclusion object is designed to import data from an external file, the instructions concerning the import operation to be performed are stored as a character string in the ATT field. If any action criteria are specified for the import operation, such criteria are stored together with the import instructions. Certain types of attribute information, however, are stored in the ATT field in the form of pointers to records in the other files of the knowledge base, the Statement Data File **42** and the Expression Data File **44**.

If possible alternative values for an object are predefined, these values are stored in the Expression Data File, which is a repository for all unique values that have been identified in the knowledge base. For example, a screen object might ask the question "What is the type of purchase contract involved?" and supply a menu of predefined choices for the user to select among. Each of these predefined choices is a possible value for the object, and each is stored in the Expression Data File. Similarly, an internal conclusion object might have four alternative values that it might conclude under different fact situations, and these possible values could be either literal strings (e.g. "YES") or evaluatable expressions (e.g. one object's value added to another object's value). Each of these strings or expressions is stored as a value in the Expression Data File.

Each unique possible value in the Expression Data File is assigned a unique, 4-digit, numeric system identifier, and this identifier is stored as a value designator in the object's ATT field, serving as a pointer to the corresponding entry in the Expression Data File. In this way, possible values may be stored only once, and may be reused by other objects in the knowledge base.

To facilitate editing the relevance and action criteria of an object, an interface is provided for the "table view" of criteria, to support the entry and manipulation of criteria statements. In this interface, each statement occupies a row on the screen, and the rows are divided into columns according to the columns in the table view (see the section entitled. 'The "Table View" of criteria'). After the developer edits and chooses to save the new version of the table view, the statement set is stored as the criteria for the object. Relevance criteria for an object are a distinct attribute in their own right, while action criteria are stored together with the data elements to which they relate.

To facilitate the reuse of statements, and of the expressions used within statements, the storage of criteria sets is accomplished through the use of pointers to the Statement Data File **42** and Expression Data File **44**. The Statement Data File is a table of unique statements that have been entered into the system, each with a unique, 4-digit identifier.

Creating the Generated Data File 14

To generate a usable system from the knowledge base that has been created, the objects in the knowledge base are organized by the Generator Program **8** into a linear sequence for processing, using a conventional topological sorting algorithm. This sequencing operation is based on identifying and taking into account all inter-object dependency relationships that have been created by the statements used in the objects' criteria sets, and by references to objects in other attributes (for example, an object value may be dynamically embedded directly into the text of another object, creating an inter-object dependency relationship without the involvement of a criteria statement). When the resulting system is executed, all objects are considered in sequence. The final sequence position of an object becomes an additional attribute of the object.

The notable feature of the resulting sequence is that all objects on which a given object depends (called its list of "upstream" objects) will precede it, and all objects which in turn depend on the given object (its "downstream" objects) will follow it. When sorting the objects into their final system sequence, they are considered in the sequence order specified by the developer. Therefore, the developer-defined sequence is respected to the extent possible. Only if an object must be moved to a different position in recognition of an unsatisfied inter-object dependency will the developer-defined sequence be disrupted. If an object must be moved, it is because it depends on some later object, and so it must be moved downward in the sequence to a position after the object on which it depends. It is inserted into the sequence position immediately following the required object.

An implication of this sequential sorting of objects is that incidences of circular reasoning may be identified by the system. If an object cannot be placed in the resulting sequence so that the sorting condition is me-

t—i.e., so that all objects on which it depends precede it, and all objects dependent on it follow it—then there is some circular element to the object dependency relationships. For example, suppose object X depends on objects A, B and C, and objects D, E an F depend on X. Now suppose that object B depends on object F. X must be sequenced after B, F must follow X, yet B must be sequenced after F. Such circular references indicate a fault in the system logic, and must be resolved before the system may successfully be generated.

In the preferred embodiment, once the objects have been sequenced, a Generated Data File **14** is generated to hold all of the necessary data about the objects and the statements. (See FIG. 6.) The addresses of those bytes in the Generated Data File where each set of necessary data about each object and each statement begin are recorded in control strings (discussion below), so that when data is required about a statement or object, the Object Processing Program **9** can go to the appropriate Generated Data File address and retrieve all necessary data. This Generated Data File is accessed at runtime using low-level file functions, which allow direct manipulation of the file pointer. Certain application-level and module-level attribute data (such as the date the file was generated and version numbers) are placed in a Header string at the beginning of the Generated Data File.

An initial state of the system is also generated, in which object firing status and statement truth values are recorded in their initial states. In general, all statements are considered to be false in the initial state, and all objects are considered not to fire. However, there are exceptions to this general rule. Some objects always fire (for example, in order to collect initial data that is always required). By convention, such objects will have no assigned relevance criteria, and the initial system state calls for these objects to fire. Some statements are true at the outset: for example, suppose a statement, referring to the value of object X, reads:

X=="" (i.e., the value of object X is the null string)

This statement is true in the initial state of the system, for object X has not yet acquired a value. The initial system state will record a truth value of "true" for this statement. The Statement Queue Array **34** (see "Statement and Object Queue Arrays" below) will be preloaded with references to these statements at the start of processing **110**.

As show in FIG. 6, the structure of the generated Data File is:

<Header information>
<Statement data>
<Object data>
<Control strings>

Statement data is represented as follows:

<the statement's identifier>
<generated code for the statement>
<list of pointers to objects that can be affected by the statement's truth value> (each pointer to an object is coupled with the sequence number of the object's designated controlling object (discussion below))

Object data is represented as follows:

<the object's identifier>
<all object attribute data, separated by attribute codes (see below)>
<list of pointers to objects that can be directly affected by the object> (each pointer to an object is coupled with the sequence number of the object's designated controlling object)
<list of pointers to statements that can be affected by the object> (each pointer to a statement is coupled with the sequence number of the statement's designated controlling object)

Control strings (see discussion below):

<Object Firing Control String>**20**
<Statement Truth Control String>**22**
<Default Value Control String>**24**
<Object Address Control String>**26**
<Statement Address Control String>**28**

Attribute codes are internal separators in an object's attribute data that mark divisions between attributes. Each consists of a delimiter, followed by a code that describes the nature of the attribute whose data follows. If the text of an object was "Enter the amount of the mortgage:", its attribute data in the Generated Data File would include the segment:

...<delimiter><code>Enter the amount of the mortgage:<delimiter><next attribute's code and data>...

The Object Processing Program 9

The inferencing mechanism works by considering each object in a defined sequence. In its simplest form, each object is examined in turn, beginning at the start of the defined object sequence, and continuing until all objects have been evaluated and the sequence ends. When an object is evaluated, its relevance criteria are evaluated to determine if the object should fire. If not, the analysis moves on to evaluate the next object in the defined sequence of objects. If the object does fire, action is taken according to its defined attributes. A question may be asked of the user, a conclusion drawn, an importation or exportation of data performed, a subroutine executed, a message displayed, or a system variable updated. The object acquires a "value", and this value is recorded, to be used in the evaluation of criteria statements employed by subsequent objects. Typically, the values that are acquired by objects are stored in a temporary buffer or data file (the Object Values File **30**), in which objects may be looked up and their values retrieved. (See FIG. 9).

Evaluation of a criteria statement occurs by treating it as a line of executable computer code, and executing it to receive a logical value in return. For example, in FoxPro, such an evaluation may be performed using the "&" operator (referred to as the "ampersand" operator or the "macro" operator). If X is a memory variable containing the character string "5>3", then the statement

```
Y=&X
```

evaluates to "true" and stores a logical value of .T. to the variable Y. Without the ampersand (i.e., if the statement read "Y=X"), the statement would simply store the contents of X to Y, i.e. Y's new value would be the character string "5>3". With the ampersand, X's contents are evaluated and the result of the evaluation is stored to Y. Alternatively, FoxPro also provides an EVAL() function, designed to evaluate expressions, and this may be used in place of "&" operator.

In this fashion, a criteria statement may be stored to the memory variable X, and X may be evaluated to

yield a logical truth value. References in the statement to objects must be provided with the current values of those objects, however, and this can be accomplished in at least two different ways.

(1) The values of the objects can be looked up in the Object Values File 30, and their literal values can be inserted into the statement in place of the object references, using string manipulation functions. For example, if the statement X takes the form "A>B", and A and B are objects, each of them would be looked up and their current values retrieved. If A's current value was 5, and B's value was 3, then the characters "5" and "3" would be inserted into the code string in place of the "A" and "B" characters. Several character string functions could be used for this work; one of these is FoxPro's STRTRAN() function, which translates instances of a given character in a string into another character:

X="A>B" (This is the initial version of the statement)

(Now retrieve current values of A and B. A=5, and B=3.) X=STRTRAN (X,"A","5") (Replace every instance of "A" in X with "5") (X's value is now: "5>B") X=STRTRAN(X,"B","3") (Replace every instance of "B" in X with "3") (X's value is now: "5>3", and now X is evaluatable with the "&" operator) Y=&X (Evaluates the contents of X, executing the assertion "5>3") Y now has the value of .T. (logical "true").

(2) An alternative method of evaluating the statement avoids such string manipulation and creates new memory variables, using the names of the objects. In the above example, when the current values of A and B are retrieved, two new variables, A and B, are created and these values are stored to the variables:

A=5
B=3

Now the content of X (the string "A>B") is directly executable:

Y=&X

FoxPro (or whatever language is being used) will recognize the references to the memory variables A and B, and will substitute their values in the evaluation of the statement, to yield the logical "true".

Because each object's relevance criteria statements are clustered with the object, the object can be considered for the first time when it is reached in the sequence. At that time, its criteria are examined and statements in its enabling pathways are evaluated, using the current values of any objects referenced in the statements. If a pathway succeeds, the object fires, and remaining pathways are not considered; if no pathway succeeds, the object does not fire and is ignored. An important consequence of this simple approach is that every object in the system will have its criteria evaluated.

In the preferred embodiment of the method, however, this simple approach of examining and evaluating each object in the sequence is made much more efficient through the use of dependency pointers and control strings. This approach allows the system to "look ahead" and efficiently propagate the implications of newly acquired information at the time such information is received. In this fashion, objects downstream in the analysis sequence can be "turned on", i.e. designated to fire in advance of reaching them in the sequence, if information acquired by upstream objects warrants that action. When system processing reaches an object in the defined sequence, its firing behavior will already have been determined.

One important implication of this approach is that not all objects need to be evaluated. In general, objects start out not firing, and if nothing occurs upstream in the analysis to change that status, the object will continue to not fire and when reached it can safely be bypassed without an explicit examination of its relevance criteria. This can lead to significant improvements in system performance, for potentially thousands of non-firing objects could be bypassed with essentially no processing.

Control Strings 60

To support this approach, control strings 60 are generated by the Generator Program 8 along with the system data to record the firing status of objects and the truth values of statements, and other required information. Several of these control strings are bitmaps (character strings consisting of ones and zeroes, such as "0011010110 . . . "), and some control strings store literal data. They are stored in Generated Data File 14.

One such control string (the Object Firing Control String 20) holds the firing status of objects in a bitmap, where a "1" means the object will fire and a "0" means it should not fire. Each object is represented in the control string as a single character, and objects are identified by their sequence numbers, which map positionally into the string. Thus if the Object Firing Control String begins with the series "1001000 . . . ", the first object in the sequence will fire, the next two objects do not fire, the fourth object fires, and the following three do not fire. This control string is constructed when the system is generated and the initial system state is described. There are as many digits in the string as there are objects; all objects are represented by zeroes, except for those objects that always fire, which are represented by ones.

A similar bitmap control string is generated for statement truth values (the Statement Truth Control String 22), in which statements that evaluate to "true" are represented by a "1" and statements that evaluate to "false" are represented by a "0". Here, since statements have no sequence, they are mapped into the control string using their unique identifying numbers. Thus, if the statement with an identifying number of "0147" evaluates to "true", then the 147th digit in the control string will be "1". When the initial system state is generated, all statements are represented with "0", except for those statements which initially are "true", which are represented with "1".

Processing in the system occurs by examining digits in the Object Firing Control String. The consultation begins by starting at the beginning of this control string, and terminates when the end of the string is reached. A pointer into the control string 112 is maintained to track the current position in the string (the object in this sequence position is referred to as the "current object"). At any point in the process, the pointer can be repositioned to a different place in the string, if desired. For example, if a user flags an earlier object for review, the system can jump back to that object and reprocess it simply by changing the value of the Object Firing Control String pointer.

In the Object Firing Control String, if the character being examined is "0", the object in that sequence position is considered irrelevant and is passed by. If the character is "1", the object fires, and may acquire a new

value. This value change may cause changes to the truth values of statements that refer to the object. Such statements are flagged for evaluation, and this evaluation occurs after each flagged statement's controlling object (see below) is processed. If a statement's truth value changes, its representation in the Statement Truth Control String will flip from a "0" to a "1" (if it goes from false to true), or from a "1" to a "0" (true goes to false).

Changes to the truth value of statements in turn affect whether other objects downstream, whose behavior depends on the truth values of these statements, will fire. Such objects are flagged for evaluation, and this evaluation occurs after each flagged object's controlling object is processed. If an object's firing behavior changes, the representation for the object in the Object Firing Control String may flip from "0" to "1" (the object now fires), or from "1" to "0" (an object that was previously designated to fire will now not fire).

A Default Value Control String **24** is also created when the system is generated. A default value may be assigned as an attribute of an object, and this value must be assigned to the object during the consultation if the object does not fire. Since an unfiring object will generally not have its attribute data examined, and in most cases will not acquire a value, these special cases of default values must be recognized. To provide this capability, a Default Value Control String is generated which duplicates the Object Firing Control String. In this case, however, any "1" characters in the string designate objects that have a default value attribute, which should be assigned to the object if the object does not fire. If an object is being bypassed because its character in the Object Firing Control String is a "0", the Default Value Control String is consulted to see if the character for this object in that string is a "1". If so, the object data must be extracted from the Generated Data File **14** and the default value assigned to the object at that time.

Addresses in the Generated Data File where object and statement data sets begin are recorded in an Object Address Control String **26** and a Statement Address Control String **28**. Object addresses map positionally into the Object Address Control String by object sequence number, i.e. the address for the 42nd object in the object sequence will be the 42nd address stored in the Object Address Control String. If the object in sequence position **42** is currently being processed, and it fires, the Object Processing Program **9** will need its data. To locate the data, the system will consult the Object Address Control String and extract the 42nd address. This is the byte address in the Generated Data File **14** for the start of this object's data. The file pointer in the Generated Data File is moved to that address, and the next sequence of data is read into memory. A delimiter marks the end of each object's data set.

Statement addresses are stored in the Statement Address Control String **28** and are coupled with statement identifier numbers. Statements are not sequenced in the system as objects are, and the Generated Data File for a given module will likely use a subset of all unique statements in the application. Therefore, the most efficient way to store statement addresses is by storing the statements' identifiers along with their addresses. Thus if a statement's identifier is "0165" the Statement Address Control String will be searched for this identifier, and the file address for the statement's data will be stored in the bytes immediately following this entry. To retrieve the statement's data, the file pointer in the Generated Data File is moved to this address and the data read into memory.

Since knowledge about object behavior is represented through the use of statements, and these statements create inter-object dependency relationships which are fully known once the knowledge base is defined, pointers can be generated by the Generator Program **8** along with other system data to indicate which statements and objects can be affected by changes in the values of particular statements and objects.

Objects and statements are treated separately in this pointer-based approach. Once the sequencing of objects has been accomplished, all inter-object dependency relationships have been identified. Similarly, it is known, from an examination of each of the unique statements used in the system, which statements can be affected by a change in the value of an object. Each object thus acquires as an additional attribute a list of all of the statements whose truth values can be affected by a change in its value. Correspondingly, each statement has an associated list of all of the objects whose behavior can be affected by a change in the truth value of the statement.

During execution of the resulting system, when an object changes value, the list of statements that can be affected by its value is consulted, and each statement in the list is flagged for a re-evaluation of its truth value. If, upon such a re-evaluation, a statement's truth value changes, the list of objects that can be affected by its truth value is consulted, and each object in the list is flagged for a re-evaluation of its firing behavior. In this manner, the consequences of newly acquired facts are propagated down the sequence of objects, causing new objects to be designated to fire when their turns come or suppressing the firing of other objects that otherwise would have fired.

When an object fires, it may have action criteria within its data which govern the behavior of attributes (for example, the criteria for concluding alternative values, or the criteria for importing a value). In these cases, the action criteria are evaluated directly when needed during the processing of the object. For example, if a conclusion object fires and it has several alternative values, the first alternative's action criteria will be examined. The action criteria statements are stored as statement identifiers, and for each such identifier, its corresponding character in the Statement Truth Control String is examined. If it is a "1", the statement is true under the current facts, and the next statement in the pathway is examined. If it is a "0", the entire pathway fails, and the next pathway will be examined. If no pathway succeeds, the value is not concluded, and the action criteria for the next alternative value in the series will be examined. Typically, the final alternative value in the series will have no action criteria assigned to it, and this value will always be assigned to the conclusion if this value is reached in the evaluation of the alternative values list.

It often happens that a user will interrupt the processing of the consultation in order to go back to a prior object to see a question or message again, and the user may change the response previously given to a question. When this happens, the consultation must proceed from that point and must reprocess objects that it processed during the first pass. The reprocessing of objects that

have already acquired values creates a special situation in the consultation.

The changed value of the upstream object will trigger the re-evaluation of affected statements. If the truth value of an affected statement changes, affected objects will be reevaluated. Objects which have already been processed and whose firing behavior is unchanged by the change in the value of the upstream object will simply be bypassed in the second pass of the consultation. If they previously fired and acquired values, these values are accepted and the consultation moves on. In certain situations, however, an object that previously fired and that still fires on the second pass should be refired anyway. For example, an object which imports data from an external file should be refired, for the change in the value of the upstream object conceivably could change the behavior of the import operation or the value of the data that is imported. To handle these situations, a system flag **90** is set in the Object Values File **30** to signal that the object should be refired.

Controlling Objects

The imposition of a sequence for objects introduces the notion of a "controlling object". A given object will have references to other objects among its attributes, in the form of statements, text embeds, lookup keys for imported data, etc. These are its set of "upstream" objects: objects on which its behavior depends. The controlling object is that object in the set of upstream objects with the highest system sequence number.

Suppose object X, in its relevance criteria, refers to five upstream objects: A B C D and E. Among those five, the last to be considered, the one with the highest sequence number is the controlling object for X's evaluation: X cannot be evaluated properly until all five objects' values are known. Any prior attempt to evaluate X would be pointless, and would create additional, wasted processing. Therefore, an evaluation of X must wait until the controlling object is reached and evaluated.

Statements have controlling objects as well. A statement may refer to various objects, one of which will have the highest system sequence number. The statement cannot be evaluated properly until all of these objects' values are known, i.e. until its controlling object is processed.

Therefore, among the attributes of an object is the list of pointers to statements which it can affect, and for each such statement, the controlling object of the statement is noted. For each statement that can be affected, a composite pointer is built consisting of the sequence number of the statement's controlling object, and the identifier of the statement to be evaluated. This composite pointer is added to the Statement Queue Array (see below) when the object's value changes, in order to flag the statement for re-evaluation.

Each statement's data, in turn, contains a list of objects which use the statement, and the controlling object of each of those objects is noted. For each such object that can be affected, a composite pointer is built consisting of the sequence number of object's controlling object, and the sequence number of the object to be evaluated. This composite pointer is added to the Object Queue Array (see below) when the statement's truth value changes, in order to flag the object for re-evaluation.

Finally, while statements can only affect objects, objects not only can affect statements but can also affect other objects directly as well. This is called a Direct Object Link. For example, where an object's value is embedded in another object's text, a change in the embedded value will affect the object's text and might create a need to redisplay it. Composite pointers for Direct Object Links, consisting of the sequence number of linked object's controlling object, and the sequence number of the linked object, are also built and included among an object's attributes if it has such links. When the value of the object changes, the composite pointers to linked objects are added to the Object Queue Array. Furthermore, the system flag **90** in the Object Values File **30** is set to force a refiring of the linkedto object when it is reached.

The design goals sought in the generation of the system and in the overall processing algorithm are to minimize generated data in the Generated Data File **14**, and to minimize computation time at execution. The truth value of a statement should be evaluated only when it needs to be: when one of its inputs has changed value and the consultation has reached the statement's controlling object. Similarly, an object's relevance criteria should be evaluated only if necessary: when the truth value has changed for one of the statements on which it depends, and the consultation has reached the object's controlling object.

Statement and Object Queue Arrays **32, 34**

In order to support the flagging of statements and objects for re-evaluation, two arrays created at the time of execution are used as queues to hold pointers to flagged entities. One array is the Statement Queue Array **34**, which contains pointers to flagged statements, and the other is the Object Queue Array **32**, which contains pointers to flagged objects.

Before leaving an object and going on to the next one in the sequence, the Statement Queue Array must be examined to determine if the current object is the controlling object for any statements in the queue. If so, those statements are to be reevaluated at this point. Similarly, before leaving an object, the Object Queue Array must be examined to determine if the current object is the controlling object for any objects in the queue. If so, those objects are to be re-evaluated at this point.

Upon acquiring a new value for an object, the object's list of all statements that use this object is examined. These statements are added to the Statement Queue Array using a composite pointer which identifies the statement's controlling object, and each will be evaluated when its controlling object is reached. In some cases, that is the object currently being processed. For example, suppose a statement reads "{OBJECT_7}>0". When OBJECT_7 is processed and acquires a value, this statement is one of the statements that can be affected by the value, so a pointer to the statement should be added to the Statement Queue Array. Since the statement depends on OBJECT_7 only, OBJECT_7 is also the statement's controlling object, so the statement should be evaluated before leaving OBJECT_7. The order of processing events is:

(1) Acquire a value for OBJECT_7
(2) Add a pointer for this statement to the Statement Queue Array
(3) Before leaving OBJECT_7, search the Statement Queue Array to see if OBJECT_7 is the controlling object for any statements. If so, evaluate those statements.

The Statement Queue Array holds pointers to statements that have been flagged because of changes in value for referenced objects. For a statement in this queue, the array holds a composite pointer containing the sequence number of the statement's controlling object, and the identifier of the statement. When that controlling object is reached, the reference to this statement is recognized and removed from the array. For efficient searching of the array, controlling objects are listed only once, at the beginning of each data element in the array. Following each unique controlling object sequence number is a list of all statements (in the Statement Queue Array) or objects (in the Object Queue Array) which have been flagged for re-evaluation when the controlling object has been processed. The array is scanned to find the controlling object of interest. If it is found, its list of statements or objects is read into memory for processing, and the data element containing the controlling object and its list is removed from the array (See FIG. 8).

The statement's identifier is used to find and extract from the Statement Address Control String the address in the Generated Data File where the statement's data begins. The file pointer in the Generated Data File is moved to that address, and the statement's data is read into memory. This data set includes the line of code that was generated for the statement by the Generator Program 8, and also the list of objects that can be affected by a change in the statement's truth value.

The generated code for the statement is extracted from this data. References to objects in a statement's generated code take the form of an "O", followed by an underscore, followed by the object's unique identifier, i.e. "O_3649". For each such instance of an object reference, the current value of the object is retrieved from the Object Values File, and a temporary memory variable is created to hold the value. This variable has the same name as the object reference in the generated code, i.e. "O_3649". When all such object references have thus been transformed into references to temporary memory variables, the statement is evaluated for truth. The resulting truth value is then compared to the current truth value for the statement in question.

If the statement's truth value has changed, the new truth value is recorded, and the statement's list of affected objects is consulted. For each such object, the object's firing status may be affected by the changed truth value of the statement. However, more efficiency can be obtained by noting that in some cases the changed truth value cannot affect the object's behavior, because of the firing bias. In the preferred embodiment, the firing bias calls for firing an object only when it's relevance criteria statements evaluate to "true". Therefore, the effect of a statement's truth value going from "false" to "true" can only be to contribute toward causing the object in question to fire. But if the object is already marked for firing (because one of its other criteria pathways has already been satisfied, for example), the object's firing status cannot change and the point is moot. In such a case, there is no point in adding a reference to the object to the Object Queue Array for re-evaluation.

Similarly, if a statement's truth value changes from "true" to "false", it will contribute toward causing the object not to fire. But if the object is already not firing, nothing can change, and again no purpose can be served by adding it to the Object Queue Array to mark it for re-evaluation.

For those objects, then, where the statement's new truth value works at cross-purposes to the object's current firing status, a reference is added to the Object Queue Array to flag the object for re-evaluation. This reference consists of the sequence number of the controlling object for the object in question, and the sequence number of the object to be re-evaluated.

A similar process occurs when the controlling object is reached for an object referenced in the Object Queue Array. The reference is recognized and removed from the Object Queue Array, and the file address for the object to be evaluated is extracted from the Object Address Control String 26. The file pointer is moved in the Generated Data File to the noted address, which marks the beginning of the object's data. This data is read into memory. Included in this data is the object's set of references to relevance criteria statements, expanded into standalone pathways, each of which if satisfied is sufficient to cause the object to fire. Each statement in a pathway is referenced by its unique identifying number. The truth value of each statement is retrieved by examining the statement's character in the Statement Truth Control String. When a false statement is encountered, evaluation of that pathway stops and an evaluation of the next pathway commences. When all of a given pathway's statements are found to be true, the object fires and evaluation stops. If all pathways have been examined without finding one that succeeded, the object does not fire. The Object Firing Control String is then updated to reflect the new firing status of the object, if the firing status changed.

In this two-tier approach to the inferencing strategy, in which the evaluation of statements and objects are separated and pointers are used to flag statements and objects for reevaluation, we see that maximum possible efficiency is attained, because:

(1) Action in the system is initiated only in response to changes in the value of an object;

(2) In response to a change in an object's value:

(a) only those statements in the system that can be affected by the change are evaluated,

(b) each such statement is evaluated only once, at the point where its controlling object is reached and therefore all of its inputs are known;

(3) In response to a change in a statement's value:

(a) only those objects whose current firing behavior in the system can be affected by the change are evaluated, taking into account not only the object's dependence on the statement, but also a comparison of the statement's current truth value and the object's current firing status,

(b) each such object is evaluated only once, at the point where its controlling object is reached and therefore all of its inputs are known.

Knowledge Base to Natural Language Interpreter 6

Natural language versions of relevance and action criteria for each object in the system are highly useful for system documentation and development. Such interpreted versions also provide an effective way to communicate the knowledge being encoded into the system to interested parties outside the development process, who may be helpful in verifying the accuracy and appropriateness of the expertise being captured.

The invention includes a Knowledge Base to Natural Language Interpreter Program 6 which is a computer program that uses object translation attributes to create natural language versions of relevance criteria state-

ments 12. For a given object, its relevance criteria are retrieved from the Objects Data File 40. Each statement's pointer is followed to the corresponding record in the Statements Data File 42, and the left-side and right-side 4-digit pointer elements of the 9-digit encoded statement are extracted. These are looked up in the Expressions Data File 44, and the expressions are retrieved. The fifth digit in the statement's 9-digit representation identifies a particular operator symbol. These elements are combined as follows to form the statement:

<left-side expression> <operator symbol>
<right-side expression>

This statement is then parsed to examine its elements. For each instance of an object reference in the statement, the referenced object is looked up in the Object Data File, and its translation attribute is extracted and substituted for the object reference in the expression. A standard natural language version is substituted for the operator symbol (for example, the phrase "is greater than or equal to" is substituted for the symbol ">=" in the statement).

Other symbols that appear in the expressions used will also have natural language alternatives used in their place. For example, "+" will become "plus", "/" will become "divided by", etc. Combinations of objects are also translated into a more natural representation. For example, "<object translation>+<another object's translation>" will become "the sum of <object translation> and <another object's translation>". Parentheses and logical operators remain unchanged. Certainty factors become "with a certainty factor of at least <certainty factor value>".

The resulting interpretation of the relevance criteria is also processed to ensure that sentences begin with capital letters, and that appropriate punctuation and connecting articles are present. Preamble text is added, referencing the translation and the object type attribute of the object whose relevance criteria is being interpreted. Standard text is also included to cover special situations, such as the case where no action criteria are specified for a possible value (i.e., it always will be assigned).

Here is an example of the process and the resulting interpretation of relevance and action criteria for an object:

From the Objects Data File:

Attribute	Value
Object name:	MGMTAPPROV
Object type:	Conclusion
Translation:	Required management approval
Relevance Criteria:	9182 ← pointers to Statements
Possible Values:	YES[illegible]0321_(0432.0543)|NO ← Data File

From the Statements Data File:

Statement ID	Statement	(encoded form, with pointers to the Expressions Data File)
9182	975310987	
0321	065430765	← (left pointer: 0654
0432	087610987	(operator: 3
0543	123454321	(right pointer: 0765

From the Expressions Data File:

Expression ID	Expression	
9753	{8163}	← object references are in curly
0654	{0246}	brackets and use object ID
0765	25000	numbers as pointers to the
0876	{0357}	Objects Data File
0987	YES	
1234	{0468} + {0680}/{5432}	
4321	{6543}	

From the Objects Data File:

Object ID:	8163
Object name:	STDTYPE
Translation:	Contract type is "Standard"
Object ID:	0246
Object name:	CONTVALUE
Translation:	Contract value
Object ID:	0357
Object name:	WARRANTINV
Translation:	Warranties are involved
Object ID:	0468
Object name:	HISTPAID
Translation:	Historical payments made
Object ID:	0680
Object name:	PENDORDER
Translation:	Value of pending orders including the current one
Object ID:	5432
Object name:	YRSHIST
Translation:	Number of years of this relationship
Object ID:	6543
Object name:	AVEANNUAL
Translation:	Average annual order value for all customers

Standard operator translations:

Number	Operator	Translation
1	=	equal to
2	#	not equal to
3	>	greater than
4	<	less than
5	>=	greater than or equal to
6	<=	less than or equal to
7	$	contained in
8	!$	not contained in

The resulting natural language interpretation 12:

The Conclusion object for required management approval will fire if:
Contract type is "Standard" is equal to YES.
A value of "YES" will be assigned to this object if:
Contract value is greater than 25000
AND either warranties are involved is equal to YES,
OR the sum of historical payments made and the value of pending orders including the current one, when divided by the number of years of this relationship, is greater than or equal to the average annual order value for all customers.
A value of "NO" will be assigned in all remaining cases.

Explanation and Diagnostic Utility 7

The invention includes an Explanation and Diagnostic Utility which allows the developer and/or the user to examine the reasoning processes at work in the system. This utility is part of both the Knowledge Base Development Program 4 and the Object Processing Program 9. A request for an explanation is usually asking one of the following questions:

Why is this question being asked?
Why is this question or conclusion relevant?
How was that conclusion drawn?
Why is this message or output being asserted as true?

In each case, what is being requested is an explanation for the behavior of a particular object, i.e. why did the object fire. A less common request, but one which can be quite illuminating for a user, is: "why did this object NOT fire? " Very few expert systems have the capability to explain why events did not happen.

To explain to the user the reasoning used in the analysis to arrive at a certain result, the Explanation and Diagnostic Utility looks to the relevance criteria of the objects involved, and constructs a natural language explanation 13 based on the relevance criteria state-

ments found. Each object in the knowledge base has a set of relevance criteria statements (except for objects that always fire, which are trivial to explain), and this set of statements constitutes a complete enumeration of how the object should behave. Therefore, the explanation process for an object that fired is simply to examine its relevance criteria, determine which statements evaluated to "true", identify the pathway(s) that succeeded, and construct the explanation using those statements. In the case of explaining why an object did not fire, each pathway is examined and the statement(s) which evaluated to "false" are identified and presented as requirements that were not satisfied, collectively causing each of the possible pathways to fail.

When a request for explanation is initiated by the user, the system retrieves the data for the object from the Generated Data File **14**. The alternative pathways for firing the object are extracted from this data, and each statement within each pathway is evaluated for truth by examining its character in the Statement Truth Control String **22**. Successful pathways are identified, as are the statements which will be used in the explanation. The data for each such statement is then retrieved and its generated code is extracted. A natural language version of the statement is created by substituting object translations for object references within the code statement, by using natural language versions of the operators used, and by presenting the current values of referenced objects to support the truth values of the statements.

For example, suppose an object fires and asks a question of the user, and the user requests an explanation for why this question is being asked. The object's data includes its relevance criteria statements. Suppose there are two alternative pathways, each consisting of a single statement. The truth values of the two statements are looked up in the Statement Truth Control String, and it is determined that the object's second pathway caused the object to fire because only the second statement evaluates to "true". The second statement's address in the Generated Data File is ascertained by consulting the Statement Address Control String **28**, and the statement's data is retrieved. Suppose that the statement's generated code reads as follows:

```
O_0245+O_3299>=100,000
```

The current values for the two objects referenced are retrieved from the Object Values File **30**. That file also records the sequence number of each object. These sequence numbers allow the object's address in the Generated Data File to be identified by consulting the Object Address Control String **26**. Each object's data is read into memory and its translation is ascertained. Suppose the translation for the object with the identifier "0245" is "the number of units manufactured last year", and the translation for object "3299" is "the number of units ordered this year", and that their current values are 65,000 units and 40,000 units, respectively. The resulting explanation would be presented to the user on screen, as follows:

> This question is asked because the number of units manufactured last year plus the number of units ordered this year is greater than or equal to 100,000.
>
> The number of units manufactured last year: 65,000
>
> The number of units ordered this year: 40,000

Object translations in the explanation text are highlighted in a different color, so that the user knows these represent other objects in the system. Each such highlighted text region behaves like a menu option. If the user selects "The number of units manufactured last year", the explanation process is repeated for object "0245", and an explanation for that object will appear on screen. This allows the user to follow the system's reasoning backward, to see what logic applied and to check the values of the objects involved.

The Explanation and Diagnostic Utility is also used at the development level in order to diagnose and debug object firing behavior during development. Developers can evaluate the behavior of any object, and by selecting displayed translations as menu options, can follow links between objects, navigating inter-object dependency relationships at will.

Knowledge Base Diagramming Utility **16**

As a supporting program for the listing of the knowledge base for documentation purposes, a program produces a diagram of all inter-object dependency relationships as a pictorial form of documentation **17**. The program examines all attributes of each object in the defined sequence to identify all objects on which it depends. A file is produced showing all objects, with connecting lines drawn between objects to represent all dependency relationships.

Example of an Application

For illustration purposes, here is a trivial expert system to evaluate the advisability of buying or selling stock. If the stock is currently owned, the system will evaluate the historical cost of the stock owned, and compare it with the current price per share. It will recommend selling the stock if a gain can be realized on the sale, else it will recommend holding the stock. If the stock is not currently owned, the system will look to see how much cash is available in the investment account, and report how many shares of the stock could be purchased with available funds. For simplicity of illustration, the possibility of buying additional shares of a stock that is currently owned is ignored, and commissions on the transactions involved are also ignored.

Each object can have numerous possible attributes. In this example, only those attributes of immediate interest are shown, for clarity of presentation.

The knowledge base will require an application record to hold application-wide and module-specific attributes:

Attributes	Values
Application number:	1
Application name:	Simple Stock Advisor
Module number:	1
Module name:	Simple Stock Advisor

In this simple application, we are calling the application as a whole Application **1**. It has one module, which is given the number **1**. In an application that had more than one module, each module would be given an individual name, and each such name would likely differ from that of the application as a whole.

Now the following objects are created:

Attributes	Values
Name:	STOCKNAME
Type:	Screen, user input
Sequence:	1
Answer length:	25
Text:	Enter the name of the stock:
Criteria:	<none>
Valid:	LEN(ALLTRIM({STOCKNAME})) > 0
Error:	You must provide the name of a stock here.
Name:	PRICENOW
Type:	Conclusion, import
Sequence:	2
Import:	MARKET.DBF/MKTNAMES.IDX/{STOCKNAME}/PRICE
Criteria:	<none>
Name:	VALIDSTOCK
Type:	Conclusion
Sequence:	3
Values:	YES■{PRICENOW} > 0 | NO
Criteria:	<none>
Name:	NOSTOCKMSG
Type:	Screen, message
Sequence:	4
Text:	I could not find a stock listed under the name: {STOCKNAME}. Please be sure you have the correct name for your stock of interest.
Criteria:	{VALIDSTOCK} = NO
Name:	STARTOVER
Type:	System
Sequence:	5
Values:	Reset
Criteria:	{VALIDSTOCK} = NO
Name:	OWNEDSHARES
Type:	Conclusion, import
Sequence:	6
Import:	PRTFOLIO.DBF/STKNAME.IDX/{STOCKNAME}/SHARES
Criteria:	<none>
Name:	OWNSTOCK
Type:	Conclusion
Sequence:	7
Values:	YES■{OWNEDSHARES} > 0 | NO
Criteria:	<none>
Name:	COSTPERSHR
Type:	Conclusion, import
Sequence:	8
Import:	PRTFOLIO.DBF/STKNAME.IDX/{STOCKNAME}/COST_SHR
Criteria:	{OWNSTOCK} = YES
Name:	TOTALCOST
Type:	Conclusion
Sequence:	9
Values:	{COSTPERSHR} * {OWNEDSHARES}
Criteria:	{OWNSTOCK} = YES
Name:	CASHAVAIL
Type:	Conclusion, import
Sequence:	10
Import:	INVEST.DBF/FACTORS.IDX/"Cash Available"/AMOUNT
Criteria:	{OWNSTOCK} = NO
Name:	SALEVALUE
Type:	Conclusion
Sequence:	11
Values:	{PRICENOW} * {OWNEDSHARES}
Criteria:	{OWNSTOCK} = YES
Name:	BUYABLE_SHARES
Type:	Conclusion
Sequence:	12
Values:	INT({CASHAVAIL}/{PRICENOW})
Criteria:	{OWNSTOCK} = NO
Name:	BUYVALUE
Type:	Conclusion
Sequence:	13
Values:	{PRICENOW} * {BUYABLE_SHARES}
Criteria:	{OWNSTOCK} = NO
Name:	GAIN
Type:	Conclusion
Sequence:	14
Values:	{SALEVALUE} – {TOTALCOST}
Criteria:	{OWNSTOCK} = YES
Name:	OWN_RECOMMEND
Type:	Conclusion
Sequence:	15
Values:	SELL■{GAIN} > 0 | HOLD■{GAIN} < = 0
Criteria:	{OWNSTOCK} = YES
Name:	NEW_RECOMMEND

-continued

Attributes	Values
Type:	Conclusion
Sequence:	16
Values:	BUY■{BUYABLE_SHARES} > 0 | DO NOTHING
Criteria:	{OWNSTOCK} = NO
Name:	RECOMMENDATION
Type:	Conclusion
Sequence:	17
Values:	{OWN_RECOMMEND}■{OWNSTOCK} = YES | {NEW_RECOMMEND}
Criteria:	<none>
Name:	BUYMSG
Type:	Screen (message), Output
Sequence:	18
Text:	The stock being considered is: {STOCKNAME} I recommend that you buy {BUYABLE_SHARES} shares of this stock at the current market price of {PRICENOW} per share. The total value of this transaction would be ${BUYVALUE}.
Criteria:	{RECOMMENDATION} = BUY
Name:	SELLMSG
Type:	Screen (message), Output
Sequence:	19
Text:	The stock being considered is: {STOCKNAME} I recommend that you sell the {OWNED_SHARES} shares of this stock that you now own, at the current market price of {PRICENOW} per share. The total proceeds from this transaction would be ${SALEVALUE}. Your cost for these shares is ${TOTALCOST}, and your resulting gain is therefore ${GAIN}.
Criteria:	{RECOMMENDATION} = SELL
Name:	HOLDMSG
Type:	Screen (message), Output
Sequence:	20
Text:	The stock being considered is: {STOCKNAME} I recommend that you hold the {OWNED_SHARES} shares of this stock that you now own. The current market price of the stock is {PRICENOW} per share. Your average cost for these shares is ${COSTPERSHR} per share, and if you sold your shares the resulting loss would therefore be ${GAIN}.
Criteria:	{RECOMMENDATION} = HOLD
Name:	NOTHINGMSG
Type:	Screen (message), Output
Sequence:	21
Text:	The stock being considered is: {STOCKNAME} I recommend that you do nothing with respect to this stock. You do not own any shares at this time, and there is no cash available in your investment account to enable a purchase of shares.
Criteria:	{RECOMMENDATION} = DO NOTHING

Narrative description of system operation

When the system is run, the first object, STOCKNAME, is processed. There are no relevance criteria specified for this object, so it always fires. It asks the user to enter the name of the stock of interest, allowing 25 characters for the stock name. A validity test is included as an attribute of the object which requires that the stock name not be blank. The test applies the ALLTRIM() function to strip leading and trailing spaces from the character string, and tests to see that the remaining string length is greater than zero. If not, the entire 25 characters were left blank and the error message is displayed, followed by another opportunity to enter a stock name (pressing the Escape key would terminate the consultation).

When a name is entered, the consultation proceeds to the next object, PRICENOW. This is an import that looks to an external data file of all stocks (MARKET.DBF), which is indexed by stock name (MKTNAMES.IDX). No action criteria are assigned to the import, so it is always attempted. The value of the STOCKNAME object is used as an index key and the system will try to locate a record for the stock of interest. (Curly brackets ("{}") always indicate a reference to an object.)

If the stock is found in the file, the value of the record's PRICE field of the MARKET.DBF database will be returned, to become the value of the PRICENOW object. If the stock is not found, the import fails, and the value returned will be zero.

The next object, VALIDSTOCK, uses the value of the PRICENOW object to determine whether the stock name entered exists. If the value is greater than zero, the import must have succeeded, and a value of "YES" is assigned to the VALIDSTOCK object. Note the format of the Values attribute for this object. Possible values are separated by the character "|". Within a value, action criteria for that particular value follow

the value itself, separated by the character "■" (ASCII 254).

Values are evaluated in the order specified in the attribute. If the value of the PRICENOW object is zero, then the import failed. When considering the first value for VALIDSTOCK, the action criteria statement for the YES value ({PRICENOW}>0) will evaluate to false, and therefore the YES value will be rejected. The next value in the series, NO, has no associated action criteria, and therefore it is always assigned. Note that the sequential evaluation of candidate values in effect implies action criteria for the NO value. If the action criteria for the earlier value are satisfied, the NO value is never considered. If it is considered, then all earlier criteria have failed, and this value covers all remaining cases.

No relevance criteria are specified for the PRICENOW and VALIDSTOCK objects, for we always want to fire them in order to ensure that we are dealing with a valid stock.

The NOSTOCKMSG object will fire only if its relevance criteria are satisfied, i.e. only if the value of the VALIDSTOCK object is equal to NO. If VALIDSTOCK=YES, then this object will simply be bypassed as inapplicable. If it fires, it displays its text as a message on the screen, and asks that the user press any key to continue after reading the message.

Note the embedded value of the STOCKNAME object in the text of this screen message object. When formatting the text for this display, the system will evaluate this embedded object reference and will substitute the user-entered stock name for the reference. The first line of the message might therefore read:

I could not find a stock listed under the name: ACME PRODUCTS.

Note also that, except where such an embedded object reference is substituted, all text in the text attribute will be displayed as formatted, including the period after the embedded reference.

If VALIDSTOCK=NO, then the next object, STARTOVER, will fire, else it will be ignored. This is a system object to initiate the "reset" action, which will release all object values and start the consultation over again from the beginning.

Note here the importance of the fact that all objects are processed in a pre-determined sequence. The remaining objects in the consultation are processed only if the consultation successfully passes the STARTOVER object: all subsequent objects may therefore assume that a valid stock name is being considered. The practical effect of this assumption is that no further references to the value of the VALIDSTOCK object are required in the relevance criteria of following objects. For example, the next object in the sequence, OWNEDSHARES, has no relevance criteria and will therefore always fire. A reasonable criteria statement for firing this and all subsequent objects might be "{VALIDSTOCK}=YES", but this is represented implicitly by the fact that the object is being processed at all.

The OWNEDSHARES object imports the number of shares of the stock currently owned from the external data file PRTFOLIO.DBF. This file is also indexed by stock name, and the value of STOCKNAME is again used to locate the appropriate record. The value of the SHARES field of the PRTFOLIO data file is returned to become the value of the OWNEDSHARES object. If the stock cannot be located in the file, the import fails and the value of this object is zero.

The OWNSTOCK object that is processed next determines whether the stock is currently owned, based on the value obtained by the OWNEDSHARES import. This object now becomes a watershed in the analysis, for the behavior of the remainder of the analysis will be affected by whether the stock is currently owned. Thus the relevance criteria of following objects will make frequent references to this object.

Note that this object in effect serves some of the functions of a node in a decision tree, causing the logic of the system to branch. If the logic at work here were to be represented as a flow chart or a decision tree, the analysis would branch one direction or the other, based on whether stock was in fact owned. This can be referred to as "pruning" the decision tree (i.e. rejecting from the analysis the paths not taken), thus reducing the search space of the analysis.

The next object, COSTPERSHR, performs another import to retrieve the historical cost per share of the shares owned. This object will fire only if the stock is in fact owned.

The TOTALCOST object that follows is a simple calculation of the total cost of the owned stock, obtained by multiplying the number of shares owned by the cost per share.

CASHAVAIL is an import from another external file, INVEST.DBF. This file, for purposes of illustration, is a hypothetical file which contains information of interest to the investor, including the amount of cash available for purchasing new stock. This object fires if the stock of interest is not currently owned: the system is checking to see if funds are available to purchase the stock. The INVEST file is a data file indexed on various descriptive phrases which might be factors of interest to the investor. The lookup index key in this case is the phrase "Cash Available", and the value of the AMOUNT field is returned.

The next object, SALEVALUE, computes the proceeds resulting from the sale of owned stock. It multiplies the current market price by the number of shares currently owned. It fires only if stock is in fact currently owned.

The BUYABLE_SHARES object divides the current market price per share of the stock into the available cash to calculate how many shares of the stock could be purchased. The INT() function is used to return just the integer portion of this calculation, to avoid fractional shares. This calculation is only performed if the stock is not currently owned. The next object, BUYVALUE, computes the total cost of such a purchase, which may be something less than the cash available if the BUYABLE_SHARES calculation resulted in fractional shares.

The next object, GAIN, computes the difference between the potential total sale value of owned stock and the total historical cost of the stock, to yield a gain (or loss, if negative) on a sale. This calculation of course only occurs if stock is owned.

Two conclusion objects now follow, OWN RECOMMEND and NEW_RECOMMEND. If stock is currently owned, the OWN_RECOMMEND object looks to the GAIN object to determine if a sale of the stock would result in a gain or a loss, and recommends SELL or HOLD accordingly. If the stock is not currently owned, the NEW_RECOMMEND object checks the value of the BUYABLE_SHARES object to see if it is possible to buy any shares at this time. If so,

it recommends a BUY, else it advises the investor to DO NOTHING.

A following conclusion object, RECOMMENDATION, collects the various possible recommendations of the system in a single object, to which later objects may conveniently point. Alternatively, these three objects could have been handled in a single RECOMMENDATION object which would cover all of the possibilities at once with somewhat more complex action criteria for the possible values:

Values:	SELL■{OWNSTOCK = YES} AND {GAIN} > 0 | HOLD■{OWNSTOCK = YES} AND {GAIN} < = 0 | BUY■{OWNSTOCK = NO} AND {BUYABLE_SHARES} > 0 | DO NOTHING

Finally, each of the four possible recommendations of the system has an associated message advising the user what to do. Various object values are embedded in these messages, in order to present appropriate information about the circumstances and possibilities. Because of their mutually exclusive relevance criteria, one and only one of these message alternatives will in fact fire.

These message objects are also designated as Output objects. The message object that fires will have its formatted text preserved, complete with embedded values, and this text can be stored to a file or printed as a page of results. Alternatively, the object's system identifier can be written to an external file, in order to identify the output of the consultation. Another system might use this result as one of its inputs. Program Control Variations

Many problems can be addressed simply by starting at the beginning of the object sequence and processing all objects until the end of the sequence is reached, whereupon the system terminates its execution. Some problems, however, require different approaches to program control. These are discussed below.

(1) Repetitive or "Batch" operations

Batch jobs, for example, are supported by providing a system reset capability and putting the system in a loop for repeated executions. In this arrangement, the system typically imports a set of data and operates on it by processing the full set of objects, taking appropriate action with the data and the results of the analysis. The system then resets itself back to its initial state, imports the next batch of data, and repeats the process until some terminating condition is satisfied to allow an exit from the loop.

(2) Progressive Refinement

In a different type of cyclical control structure, the system performs successive passes on the same set of data, refining it each time, until some terminating condition is reached. The first pass through the object list allows a high-level or "coarse-grained" manipulation of the data. Each subsequent pass takes the results of the previous pass as its input, performing more "fine-grained" manipulation.

Such cycles may also be built into a segment of what is otherwise a "single-pass" set of objects. This is supported by system "go-to" objects which, in the absence of a specified terminating condition (which might be a logical value, end-of-file condition in some external data file, or a specified number of iterations), reset the object counter back to some earlier object in the Object Firing Control String 20. This creates in effect a "do-while" looping construct within the otherwise linear set of objects.

(3) Modules calling other modules

Support is also provided for systems to call other expert systems as modules to perform sub-tasks, creating new values or updating the values of objects that have already been processed, whereupon program control returns to the calling module. In the current implementation, the system being called must be another module of the same application, because object names and identifiers may be duplicated in different applications, but are not allowed to be duplicated within modules of the same application.

A system object in the calling module performs the call to the other system. In the process, the current state of the calling module is saved, by saving to a file on disk important memory variables, including the current object counter 112, the Statement and Object Queue Arrays 34, 32 and the Control Strings 20–26. The Generated Data File 14 for the calling module is closed, but the Object Values File 30 remains open and unchanged for use by the called module. However, a copy is created of the Object Values File as it exists at the time of the call, for use when program control returns. Processing then begins for the called module, which opens its own Generated Data File and proceeds with its consultation 15. The called module may in turn call another module, and there is no limitation on the degree to which such inter-module calls may be nested.

Objects are reused within an application by being members of more than one module. If an object that is common to the two modules has been processed by the calling module, and is therefore found by the called module in the existing Object Values File, its value is accepted and used in the called module's consultation. The value of a common object may also be updated as a result of the called module's analysis. Note that almost all attributes of a common object may differ in the two modules. In particular, the firing behavior of the object may differ, because different relevance criteria and inter-object dependency relationships may be specified in the called module's data. Other attribute data of common objects that is stored in the Object Values File will be updated according to the attribute specifications of the called module, as part of its normal processing.

For example, suppose an object is a screen object in the calling module, and it fires and acquires the value "XYZ" In the called module, this same object might be a conclusion object. If it fires in the called module, its initial value in that context is still "XYZ", but the alternative values for the object as a conclusion will be processed, perhaps changing its value to "ABC". The called module will update the Object Values File to reflect this new value, and will also update the object type attribute in that file to note that this object is a conclusion.

After processing is complete in the called module, program control returns to the calling module. At that time the calling module's Generated Data File is reopened, and the Statement and Object Queue Arrays, Control Strings and other memory variables for the calling module are reinstated from the file saved to disk. A comparison is then made between the Object Values File and the copy of the file that was made at the time

of the call. The values of objects in the file are not disturbed, for typically the reason for calling the module in the first place was to acquire different values for one or more common objects as a result of the called module's processing. Other attribute data that is stored in the Object Values File, however, is restored to its original state, to match the attribute specifications of the calling module. In the example just given, the value of the common object upon return to the calling program will be its updated value, "ABC". The object type attribute stored in the Object Values File, however, is changed back to reflect the fact that this object is a screen object in the calling module, rather than a conclusion.

Processing then resumes normally in the calling module. The Object Values File will now contain updated values for objects that are common to the two modules, and will also contain additional records for objects that are members of the called module only. These extra object records are ignored in the subsequent processing of the calling module and do not affect its analysis in any way.

When another module is called in this manner and updates the value of objects common to both modules in the Object Values File, followed by a return of control to the calling module, the practical effect is a "blackboard" system of cooperating expert systems which share knowledge through the values of the common objects.

(4) Event-driven applications

Monitoring, reactive, and other event-driven applications are supported by establishing conceptual "threads" in the overall list of objects, based on inter-object dependency relationships. A thread is a continuous sequence of objects that forms a conceptually coherent subset of the overall object list, and constitutes a section of the Object Firing Control String 20. An object at the beginning of a thread is called the "thread initiator" object, and the final object in the thread is the "thread terminator" object.

An external monitoring routine waits for certain conditions to arise, then resets the object counter to the initiator object for the appropriate thread. The system then processes objects normally for the length of the thread, observing object sequence, firing applicable objects and taking appropriate actions, and after processing the terminator object of the thread, returns control to the monitoring routine. Such an design, for example, might be appropriate for monitoring pressure valves in a hydraulic system. Threads could be allowed to overlap, where objects are common to more than one thread. Such an arrangement is possible because objects are ordered and processed according to their inter-object dependency relationships, as determined by their relevance criteria.

Certainty Factors

Certainty factors qualify the applicability of criteria and object values by providing a measure of likelihood or confidence that a given condition is true, thereby supporting to some extent reasoning under conditions of uncertainty. A system supporting the use of certainty factors, for example, might conclude a given fact's value with an 80% level of confidence, and another fact's value with only a 50% level of confidence. Several algorithms exist for combining certainty factors when such factors interact, to produce a new, composite certainty factor for a resulting value.

Many, perhaps most, expert system applications do not require certainty factors. When solving problems, experts use them in some situations to evaluate degrees of belief or likelihood. Most of the time, however, the algorithms involved in combining certainty factors are too complex for an expert to use in practice, and the mere fact that uncertainty exists becomes an additional fact that is evaluated with 100% confidence together with other facts in the analysis. Also, expert systems are frequently designed not to evaluate and deal with uncertainty, but to eliminate it. Thus an expert system application in a business setting may enforce company policies predictably and uniformly by encoding the requirement that "if this condition of uncertainty exists, then always do the following".

This reveals an important distinction when thinking about reasoning under conditions of uncertainty. Uncertainty can be represented directly in a system and reasoned about with no uncertainty in the reasoning process, or it can be represented indirectly by encoding knowledge as if it were certain and then adjusting the reasoning process to reflect uncertainty about the applicability of the encoded knowledge.

For example, compare the assertion "There is a 60% chance of rain tomorrow", asserted with a 100% confidence level, with the assertion "It will rain tomorrow", asserted with a 60% confidence level. The first assertion represents uncertainty directly, and there is no ambiguity about it. Other elements in the system may use it and reason about it with full confidence that it applies. The second assertion represents uncertainty indirectly, by making an unambiguous statement, but the assertion is qualified with a certainty factor to indicate that it may not be correct.

The invention supports both alternatives for representing uncertainty. The first method relies on the inherent meaning of an object to represent uncertainty, while firing the object under specified conditions with 100% confidence that the object should be fired. A conclusion object may fire and conclude that "X is uncertain", and the system can then use that fact in its analysis. Knowledge about what the system should do if this object is applicable, i.e. under conditions where "X" is uncertain, is encoded in the relevance and action criteria of other objects, whose behavior may be affected by this condition.

The second method for representing uncertainty employs certainty factors directly, in several contexts of the knowledge base. Certainty factors can be assigned directly (e.g., "80%"), or may be derived by evaluating an expression.

(1) A certainty factor can be assigned as an attribute of an object, and when the object fires and acquires a value, this certainty factor is stored in the Object Values File 30.

(2) Certainty factors can be assigned to the values that conclusion objects may acquire, to say in effect: "under these conditions, fire this object and give the resulting value the following certainty factor". This allows different certainty factors to be assigned to different alternative values of the same object.

If a given value should be concluded with a different certainty factor under different conditions, the value is repeated as a separate alternative value for the object, with its own dedicated set of action criteria. The resulting effect is to say: "under these conditions, conclude this value for the object with a 90% confidence level,

but under these conditions conclude that same value with only a 60% confidence level".

(3) Criteria statements can acquire certainty factors as a result of the combined certainty factors of objects referenced in them, and each statement in a set of criteria can be assigned a certainty factor threshold, to say in effect: "in order to evaluate to true, this statement must have a certainty factor of at least 75%". The same statement could be re-used in the context of another object's criteria, using a different certainty factor threshold.

(4) A statement may make an explicit evaluation of an object's certainty factor by employing the CF() function, a function in the Object Processing Program, which returns an object's certainty factor, as recorded in the Object Values File. For example, a statement might read:

```
CF (OBJECT_12)>.8
```

This statement would evaluate to "true" if the certainty factor of OBJECT_12 is greater than 80%.

Validation Table Generator

Inter-object dependency relationships also allow a means of verifying and demonstrating that an expert system is behaving correctly, i.e. consistently with its defined relevance criteria. Because the criteria represent and provide for all factors which can affect an object, and since the firing behavior of objects can be measured, validation tables and test cases can be constructed to monitor and prove the behavior of each object in the system.

For this purpose, the invention includes a Validation Table Generator 5. In a validation table created by this program 11, the table itself tests one enabling pathway of an object. Each of the factors that can affect the behavior of the object are listed down the left side of the table (see FIG. 12). Each column of the test table contains a fact situation involving these factors, created by a particular test case, which is given a test case number at the top of the column. The table is divided into a left-hand portion which tests the failure of the object to fire, and a single, final column on the right-hand side which tests the firing of the object. In each of the left-hand columns, all factors of the pathway should be enabled except for one (a different factor in each column should be disabled). Since all of the factors represented within a pathway for an object must be satisfied in order for the pathway to succeed, in the test cases for each of these columns, the object should not fire, and it should not fire because one and only one of the factors is disabled. In the final column of the table, all factors should be enabled, and the object should fire.

Since a given test case will involve the processing of all objects in the system, and only a few objects will affect a given pathway, test cases may be re-used from table to table. The result of creating tables for every pathway of every object in the system will be a minimized collection of test case scripts which can be fed into the system, and the firing behavior of objects can then be checked against the behavior predicted by the test tables. In this fashion, mistakes in the specification of criteria can be checked for, and the correct operation of the system can be demonstrated empirically. Because the relevance criteria are stored in the knowledge base, the creation of test tables and test cases, and the processing of the test case scripts, can all be automated.

Alternative Embodiments

Three alternative embodiments of the invention have been devised, in addition to the preferred embodiment disclosed above. These are alternatives for the organization and implementation of the system which is generated from the knowledge base, and which actually performs the expert system's function. Each is described below.

(1) Code generation

A conventional computer program is generated, in which statements of computer code are generated for each object in the knowledge base, in the order of the defined object sequence. The main body of the program consists of the objects' relevance criteria, separated into distinct enabling pathways, and constructed as a series of nested "if" statements built from the criteria statements. Thus if a pathway for an object consisted of three statements (A, B and C), then the generated code would read:

```
IF <statement A>
   IF <statement B>
      IF <statement C>
         <statements to fire the object>
      ENDIF
   ENDIF
ENDIF
```

If a particular "if" statement fails, program control will drop to the end of this block of nested statements, bypassing further evaluation of statements in that pathway.

Object values, when acquired, are stored in memory variables using the object name as the name of the variable. Lines of generated "if" statements use these same names to refer to the object's value. The result is that each "if" statement is directly executable, because memory variables exist for each object statement. All memory variables to hold object values are created and initialized at the start of processing, in order to provide for references to objects that do not fire.

The data type of each object's expected value must be considered when generating this code and when initializing object memory variables. Variables for objects containing character data are initialized to the null string (""), and generated code for character string comparisons must embed quotation marks correctly. For example, a line of code might be generated as:

```
IF OBJECT_19="YES"
```

A memory variable named OBJECT 19 will be created at system startup and will be initialized to the null string. Note also the quotation marks around the literal string YES.

Similarly, variables for objects of numeric data type are initialized to zero, logical data types initialized to false, and date types initialized to a null date (" / / ").

Because code is generated in the order of object sequence, references to other objects' values in a given object's generated code will be meaningful. The referenced objects will have been processed, and their values acquired, prior to the program's execution of code which references their values. Default values may be assigned either initially, or upon passing the object's generated code. If the set of nested "if" statements all succeed, then program control will reach the code gen-

erated at the heart of the code block. These statements will initiate appropriate action for the object's firing. If action criteria are involved for the object—for example, criteria governing the importation of data, the appearance of a given option on a menu, or a conclusion's value—appropriate subroutines containing similar generated code are called. If the object is a screen object, a general purpose procedure is called to handle the screen interface and acquire a value for the object. Similarly, general-purpose subroutines are called to handle the concluding of values for internal conclusion objects, the posting of messages to the user, and the writing out of data to an external file.

Code is also generated to handle requests by a user to interrupt processing, and to handle other common events in the processing of the system.

In practice, this implementation can be extremely fast, but it also encounters several difficulties. The amount of generated code for a large system can be exceptionally large, and may be too large to fit in a processor's available memory, causing system crashes. This is largely because no reuse of statement code is possible: wherever a statement is used in the system, its line of generated code must be reproduced, rather than looking to a common pool of statement code in which each statement's generated code is stored only once.

Furthermore, since every object is associated with a memory variable, and the contents of these variables can in some cases be lengthy strings of text, total system memory resources can be exhausted in the course of processing.

Finally, effective program control in such an implementation is difficult to achieve, as illustrated in two contexts. First, a common situation in expert systems is that a user will wish to go back to an earlier object and change an answer. To accommodate this situation in this implementation, the processing of objects must be interrupted, program control must return to an outside loop, processing must begin again at the top of the main body of the program, processing must then bypass all objects preceding the object of interest, yet stop at the appropriate object and reprocess it. This extra processing can take a long time, and considerable code must be generated around the basic blocks of nested "if" statements in order to handle such situations correctly.

Another context which presents a difficult problem for program control has to do with the nature of the problem being solved. Problems involving the monitoring of situations and reactive responses do not lend themselves to such predetermined, procedural code. In such cases, the system must be flexible enough to go directly to an appropriate set of objects and process them as a subset of the whole knowledge base, and then return to its monitoring state. In the preferred implementation, program control can be directed simply by assigning a new object sequence number to the variable that holds the number of the object currently being processed, and processing then resumes from that object.

(2) Data-driven analysis

In another embodiment, data rather than code is generated. This data is stored as records in a file (the "master" file) in a relational database on a fixed disk, and processing occurs by driving through this data until the end of the data file is reached. Data is organized in object sequence. Each data record consists of an encoded criteria statement and pathway information. Within each object, statements are organized by pathways and by statement position within pathways. A composite index key is built, holding the object sequence number, criteria path number, and statement position number within each pathway.

When driving through the master file, the beginning of a new object is evidenced by a change in the object sequence number. Consideration begins of the object's first criteria pathway. Each statement record is examined in turn, until a new pathway number is encountered If a statement evaluates to "false" records in the file are skipped until a new pathway or new object is encountered. Each statement record contains a statement identifier. A relation is set using this identifier into another file (the "resource" file), indexed by statement identifier, in which the generated code for each statement is stored. As each statement in the master file is evaluated, the line of generated code for the statement is retrieved from the resource file and evaluated for truth. Object values are stored in an Object Values Data file, and are retrieved and used as needed in order to evaluate statements.

A relation is also set from the master file into the objects file which holds all objects and their attributes, using the statement identifier as an index and relational key. If a pathway succeeds, general-purpose subroutines are called to extract the appropriate attributes from the objects file and process the object.

Criteria evaluation in this implementation is thus transformed from an evaluation of generated "if" statements to an examination of data records in a data file. Memory problems, for both program size and for the storage of object values in memory variables, are greatly eased. Statement code is reused, in that it is stored only once, in the resource file, and all statement references by master file records are in the form of pointers. Further, more program control is gained, because the record pointer in the master file can be directly manipulated, which cannot be accomplished with the execution of conventional program code in the first implementation. For example, to go back to an earlier object and reprocess it, the record pointer is simply repositioned in the file to the start of the appropriate object's records, and processing resumes from that point.

Despite these advantages, however, this implementation can suffer from predictable performance problems. In a large application, large numbers of data records will be generated, and system performance can be hampered by the frequent interactions with the disk drive that are required in order to access the data. Performance also suffers from the continual need to update relational pointers among the different data files. These problems may be addressed to some extent through the use of faster and more powerful hardware, and the use of disk caching, but they cannot be eliminated altogether.

(3) Hybrid approach: Generated data and generated subroutines

This approach uses a combination of the above two implementations. Program control is handled by the use of data files, as in the second implementation, but for the consideration of object criteria, generated subroutines are called. For each object during system generation, a dedicated subroutine is generated which holds the code for the statements to be evaluated, and which returns a result indicating whether the object should fire, based on the execution of the generated code. Also, some attribute data is assigned by the subroutine to common

system memory variables. (For example, the system memory variable T would hold the text of the current object being processed.) The name of the object's subroutine is some variant of the object's name. During processing, object records in a data file (one record per object) are examined in object sequence, and their appropriate subroutines are called to determine if the object fires, and to acquire necessary attribute data.

This implementation succeeds in combining the best features of the first two implementations. Program control is maintained by considering objects in a data file, and repositioning the record pointer as needed. Performance is facilitated by using generated "if" statements rather than data records for criteria evaluation, and assigning attribute data to variables in a procedure, to avoid retrieval of such data from the objects file.

Despite these advantages, some performance problems remain. Large amounts of program code are still generated in the dedicated object subroutines, for again no reuse of statement code is possible. The system still has potentially large data files, with resultant hardware performance limitations, and a relational pointer from the master file into the objects file is still necessary for some attribute data.

Finally, in all three alternative implementations, the advantages of the preferred embodiment's pointer-based approach are not enjoyed. All objects are evaluated at the time they are reached in the processing sequence. In the preferred implementation, only those objects and statements which can be affected by acquired values are evaluated, resulting in significant improvements in efficiency.

I claim:

1. A computing system including a processor and a data set, the data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true; and

(d) in one or more of the objects, the action of the object instructs the processor to set the value of the object to one of the plurality of possibilities, and,

(e) in one or more of the objects, the relevance criterion includes a reference to the value of another object such that the relevance criterion will or will not be satisfied depending on the value of the other object.

2. The computing system of claim 1 in which the action of an object comprises making a change to another object.

3. The computing system of claim 2 in which the change to another object is a change in the relevance criterion of the other object.

4. The computing system of claim 3 wherein the data set has a specified sequence of the objects beginning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence.

5. The computing system of claim 4 in which the change to another object changes the sequence of the objects.

6. A computer method for performing a computation, by use of a processor, on a data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true;

the method having steps comprising:

(d) determining the value of a first object,

(e) evaluating the relevance criterion of a second object, which relevance criterion includes a dependency on the value of the first object, and

(f) if the relevance criterion of the second object is satisfied, executing the action specified by the object and setting the value of the object.

7. The computer method of claim 6 in which the action of an object comprises making a change to another object.

8. The computer method of claim 7 in which the change to another object is a change in the relevance criterion of the other object.

9. The computer method of claim 8 wherein:

the data set has a specified sequence of the objects beginning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence.

10. The computer method of claim 9 in which the change to another object changes the sequence of the objects.

11. A computing system including means for creating or changing a data set which may be processed on a computing system with a processor, the data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true; and

(d) in one or more of the objects, the action of the object instructs the processor to set the value of the object to one of the plurality of possibilities, and,

(e) in one or more of the objects, the relevance criterion includes a reference to the value of another object such that the relevance criterion will or will not be satisfied depending on the value of the other object.

12. The computing system of claim 11 in which the action of an object comprises changing the relevance criterion of another object.

13. The computing system of claims 12 wherein the data set has a specified sequence of the objects begin-

ning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence and the action of one or more objects changes the sequence of the objects.

14. A computing system for generating validation tables from a data set which may be processed on a computing system with a processor, the data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true; and

(d) in one or more of the objects, the action of the object instructs the processor to set the value of the object to one of the plurality of possibilities, and,

(e) in one or more of the objects, the relevance criterion includes a reference to the value of another object such that the relevance criterion will or will not be satisfied depending on the value of the other object;

the computing system comprising:

(f) instructions for reading the contents of an object, and

(g) instructions for generating a validation table from the contents of the object.

15. The computing system of claim 14 in which the action of an object comprises changing the relevance criterion of another object.

16. The computing system of claim 15 wherein the data set has a specified sequence of the objects beginning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence and the action of one or more objects changes the sequence.

17. A computing system for interpreting into natural language the contents of a data set which may be processed on a computing system with a processor, the data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true; and

(d) in one or more of the objects, the action of the object instructs the processor to set the value of the object to one of the plurality of possibilities, and,

(e) in one or more of the objects, the relevance criterion includes a reference to the value of another object such that the relevance criterion will or will not be satisfied depending on the value of the other object;

the computing system comprising:

(f) instructions for reading the relevance criterion and action of one of the objects, and

(g) instructions for interpreting the relevance criterion and action into natural human language, and

(h) instructions for adding one or more natural human language words to make one or more complete natural human language sentences.

18. The computing system of claim 17 in which the action of an object comprises changing the relevance criterion of another object.

19. The computing system of claim 18 wherein the data set has a specified sequence of the objects beginning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence and the action of one or more objects changes the sequence.

20. A computing system for examining a data set which may be processed on a computing system with a processor, the data set comprising a plurality of objects, each object having:

(a) a data field which may contain a value, the value of the object, which value may be set to one of a plurality of possibilities by the processor,

(b) a relevance criterion which may be evaluated by the processor to yield a logical true or a logical false, and

(c) an action, in the form of processor instructions or a pointer to processor instructions, which will be executed by the processor if evaluation of the relevance criterion by the processor produces a logical true; and

(d) in one or more of the objects, the action of the object instructs the processor to set the value of the object to one of the plurality of possibilities, and,

(e) in one or more of the objects, the relevance criterion includes a reference to the value of another object such that the relevance criterion will or will not be satisfied depending on the value of the other object;

the computing system comprising:

(f) instructions for reading information from the data set, and

(g) instructions for determining what steps a computer would take upon execution of the actions specified by one or more of the objects.

21. The computing system of claim 20 in which the action of an object comprises changing the relevance criterion of another object.

22. The computing system of claim 21 wherein the data set has a specified sequence of the objects beginning with a first object and ending with a last object, in which sequence the relevance criterion of each object makes no reference to the value of a later object in the sequence and the action of one or more objects changes the sequence.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,418,888
DATED : May 23, 1995
INVENTOR(S) : John L. Alden

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

On title page, item [54] and Col. 1, line 1,

In the title delete

"SYSTEM FOR REVELANCE CRITERIA MANAGEMENT OF ACTIONS AND VALUES IN A RETE NETWORK" and replace with --SYSTEM FOR RELEVANCE-ACTIONS-VALUE BASED PROGRAMMING--.

Signed and Sealed this

Sixteenth Day of January, 1996

Attest:

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US006058206A

United States Patent [19]

Kortge

[11] Patent Number: 6,058,206

[45] Date of Patent: May 2, 2000

[54] **PATTERN RECOGNIZER WITH INDEPENDENT FEATURE LEARNING**

[76] Inventor: **Chris Alan Kortge**, 6432 Williams Ridge Way, Austin, Tex. 78731

[21] Appl. No.: **08/980,838**

[22] Filed: **Dec. 1, 1997**

[51] **Int. Cl.**[7] **G06K 9/62**

[52] **U.S. Cl.** **382/159**; 382/157; 706/16; 706/25; 706/30; 706/20

[58] **Field of Search** 382/157, 158, 382/159, 156; 706/20, 16, 25, 30

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,060,278	10/1991	Fukumizu	382/159
5,251,268	10/1993	Colley et al.	382/159
5,359,700	10/1994	Seligson	395/24
5,422,981	6/1995	Niki	395/22
5,568,591	10/1996	Minot et al.	395/22
5,754,681	5/1998	Watanable et al.	382/159
5,812,992	9/1998	Vries	706/25
5,822,742	10/1998	Alkon et al.	706/20
5,835,633	11/1998	Fujisaki et al.	382/159
5,870,493	2/1999	Vogl et al.	382/195
5,870,828	2/1999	Yatsuzuka et al.	706/25

OTHER PUBLICATIONS

Oct. 1990 Földiák "Forming Sparse Representations by Local Anti-Hebbian Learning", Biological Cybernetics.

Dec. 18, 1991 Schmidhuber "Learning Factorial Codes by Predictability Minimization", Univ. of Colorado Dept. of Computer Sci. TR-CU-CS-565-91.

Jul. 1996 Jaakkola & Jordan "Computing Upper and Lower Bounds on Likelihoods in Intractable Networks", in Proceedings of the Twelfth Conference on Uncertainty in AI.

Oct. 1992 Neal "Connectionist Learning of Belief Networks", Artificial Intelligence 56, pp. 71–113.

Dec. 1996 Lewicki & Sejnowski "Bayesian Unsupervised Learning of Higher Order Structure", Advances in Neural Information Processing Systems 9 (Proceedings of the 1996 Conference, Dec. 2–5).

May 1986 Rumelhart et. al. "Learning Internal Representations by Error Propagation", Parallel Distributed Processing vol. 1, MIT Press, Cambridge, MA.

Nov. 1994 Hastie et. al. "Learning Prototype Models for Tangent Distance", Advances in Neural Information Processing Systems 7 (Proceedings of the 1994 Conference, Nov. 28–Dec. 1.

Aug. 1990 Kortge, "Episodic Memory in Connectionist Networks", Proceedings of the Twelfth Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Associates, Hillsdale, NJ.

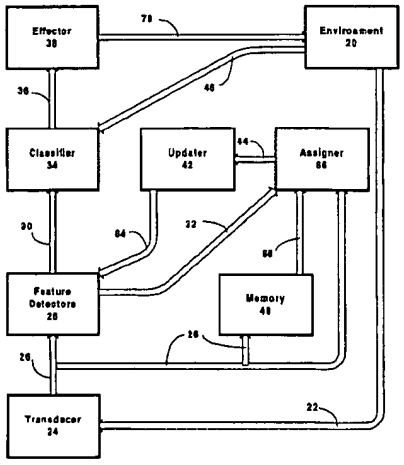
Primary Examiner—Amelia Au
Assistant Examiner—Jingge Wu
Attorney, Agent, or Firm—Taylor Russell & Russell, P.C.

[57] **ABSTRACT**

A pattern recognition device having modifiable feature detectors (**28**) which respond to a transduced input signal (**26**) and communicate a feature activity signal (**30**) to allow classification and an appropriate output action (**70**). A memory (**40**) stores a set of comparison patterns, and is used by an assigner (**66**) to find likely features, or parts, in the current input signal (**26**). Each part is assigned to a feature detector (**28**[*m*]) judged to be responsible for it. An updater (**42**) modifies each responsible feature detector (**28**[*m*]) so as to make its preferred feature more similar to its assigned part. The modification embodies a strong constraint on the feature learning process, in particular an assumption that the ideal features for describing the pattern domain occur independently. This constraint allows improved learning speed and potentially improved scaling properties.

A first preferred embodiment uses a group of noisy-OR type neural networks (**50**) to implement the feature detectors (**28**) and memory (**40**), and to obtain the parts by a soft segmentation of the current input signal (**26**). A second preferred embodiment maintains a lossless memory (**40**) separate from the feature detectors (**28**), and the parts consist of differences between the current input signal (**26**) and comparison patterns stored in the memory (**40**).

20 Claims, 12 Drawing Sheets



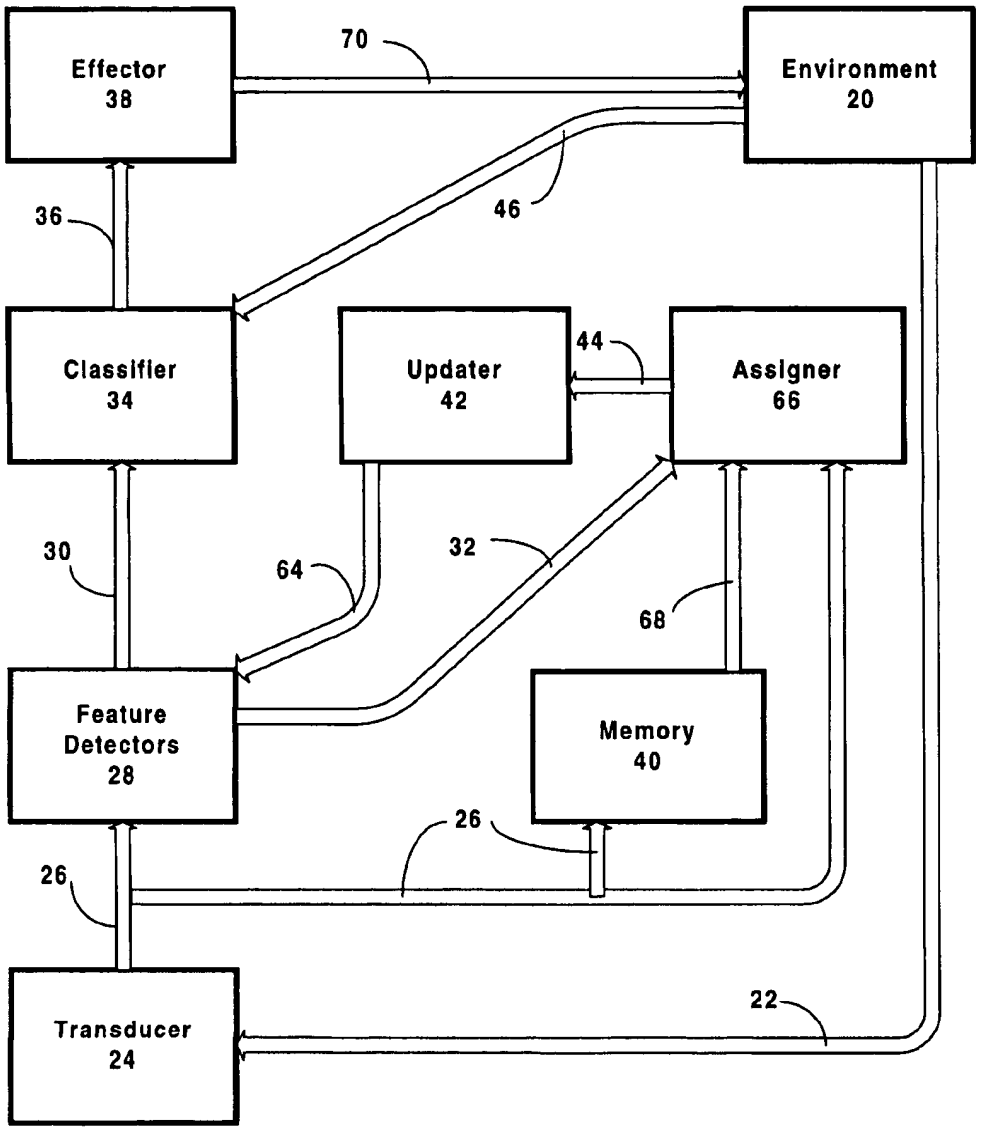

Effector
38
70
Environment
20
46
36
Classifier
34
Updater
42
44
Assigner
66
30
64
32
68
Feature
Detectors
28
Memory
40
26
26
Transducer
24
22

Fig. 1

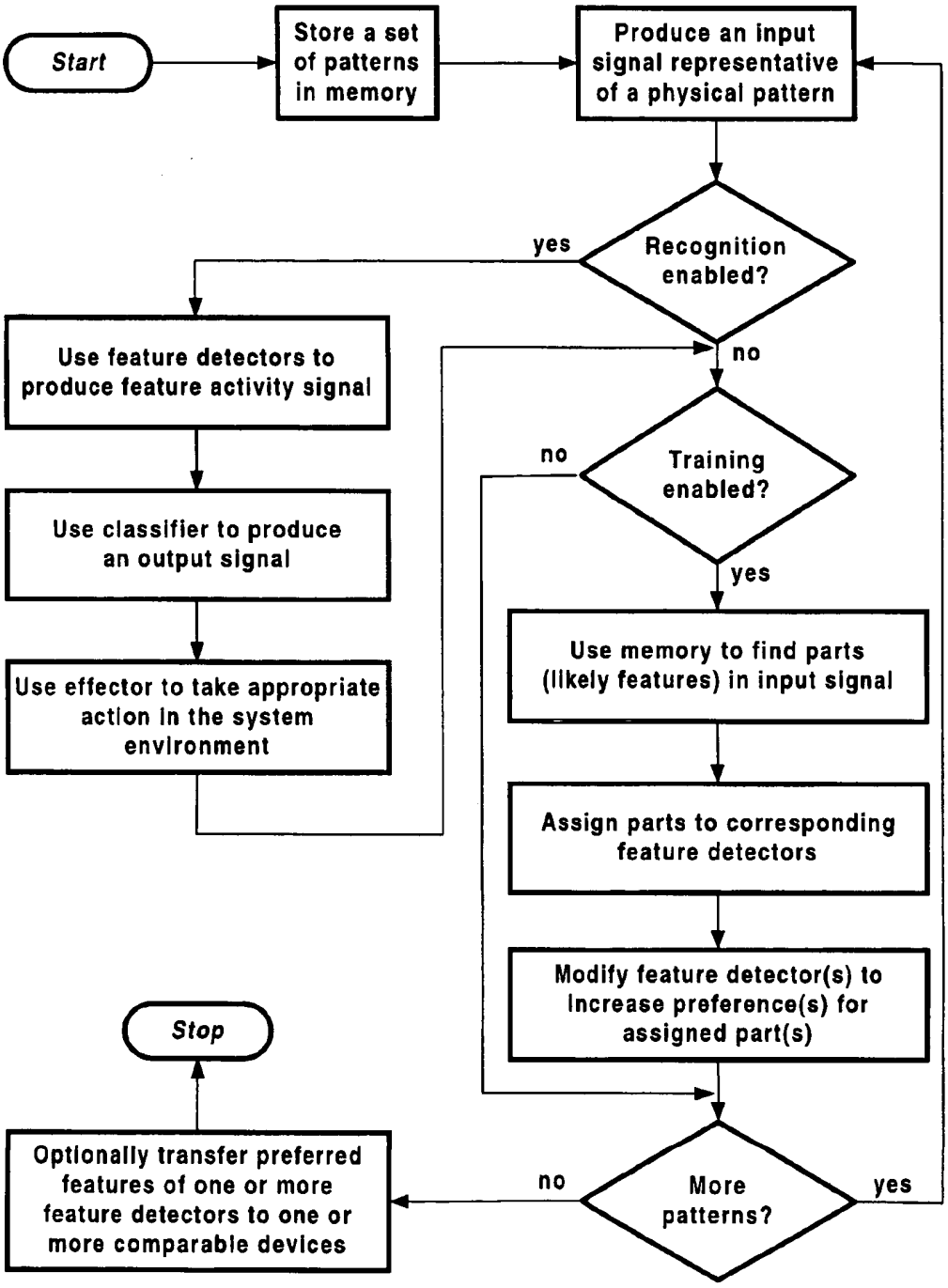

Start
Store a set of patterns in memory
Produce an input signal representative of a physical pattern
Recognition enabled?
yes
no
Use feature detectors to produce feature activity signal
Use classifier to produce an output signal
Use effector to take appropriate action in the system environment
Training enabled?
no
yes
Use memory to find parts (likely features) in input signal
Assign parts to corresponding feature detectors
Modify feature detector(s) to increase preference(s) for assigned part(s)
More patterns?
no
yes
Optionally transfer preferred features of one or more feature detectors to one or more comparable devices
Stop

Fig. 2

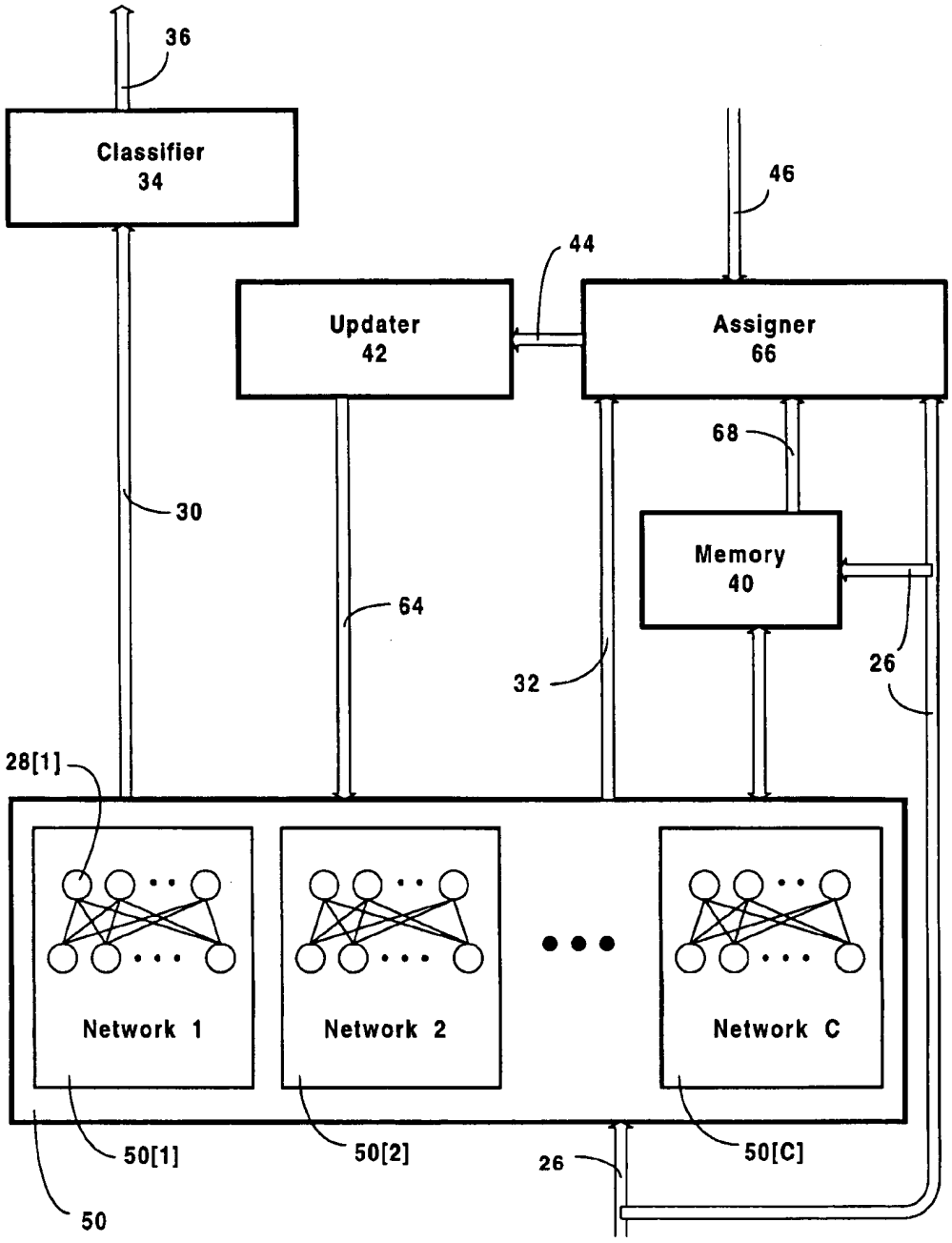

36
Classifier
34
46
44
Updater
42
Assigner
66
68
30
Memory
40
64
32
26
28[1]
Network 1
Network 2
Network C
50[1]
50[2]
50[C]
26
50

Fig. 3

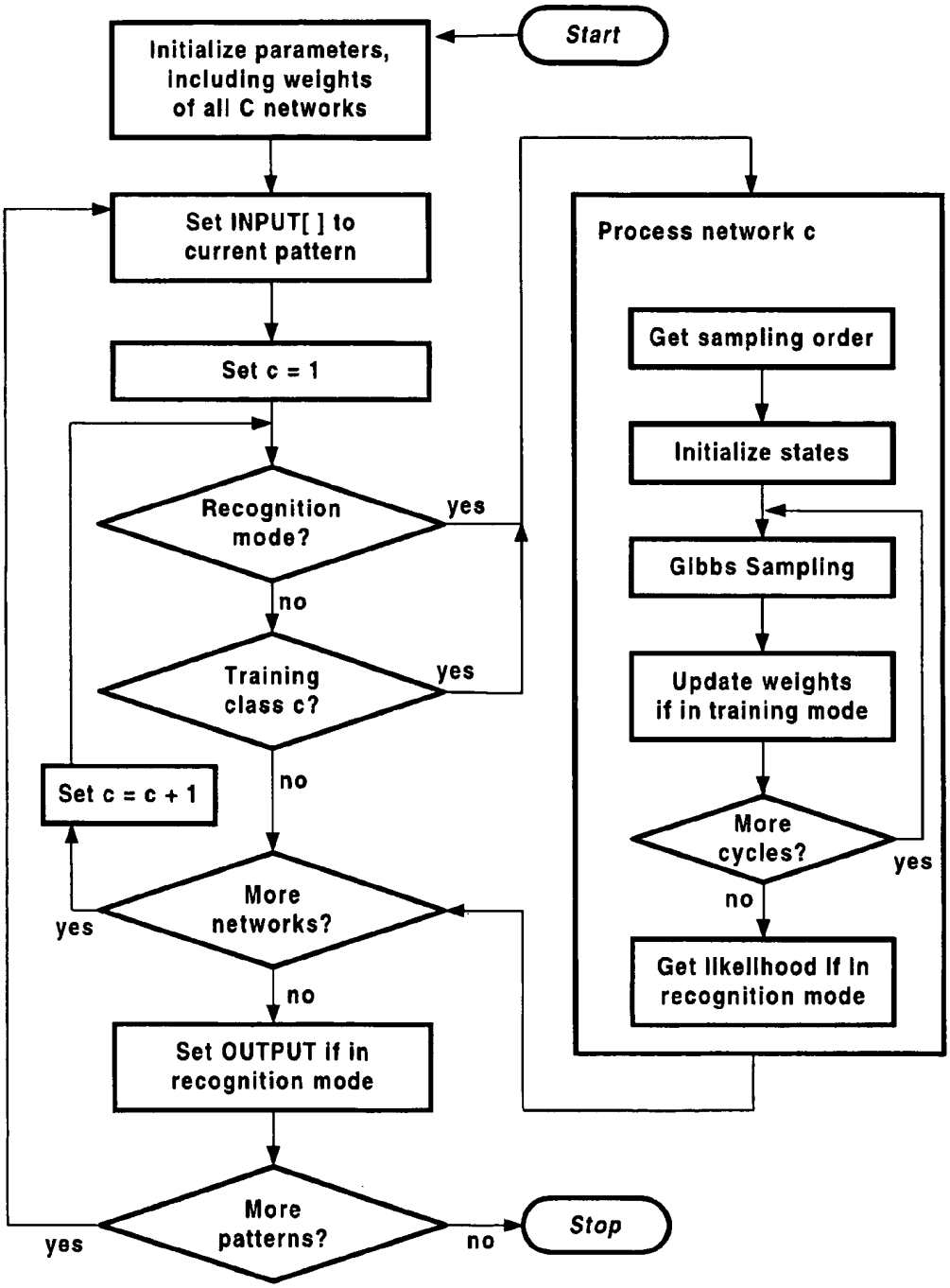

Start
Initialize parameters, including weights of all C networks
Set INPUT[ ] to current pattern
Set c = 1
Recognition mode?
yes
no
Training class c?
yes
no
Set c = c + 1
More networks?
yes
no
Set OUTPUT if in recognition mode
More patterns?
yes
no
Stop
Process network c
Get sampling order
Initialize states
Gibbs Sampling
Update weights if in training mode
More cycles?
yes
no
Get likelihood if in recognition mode

Fig. 4

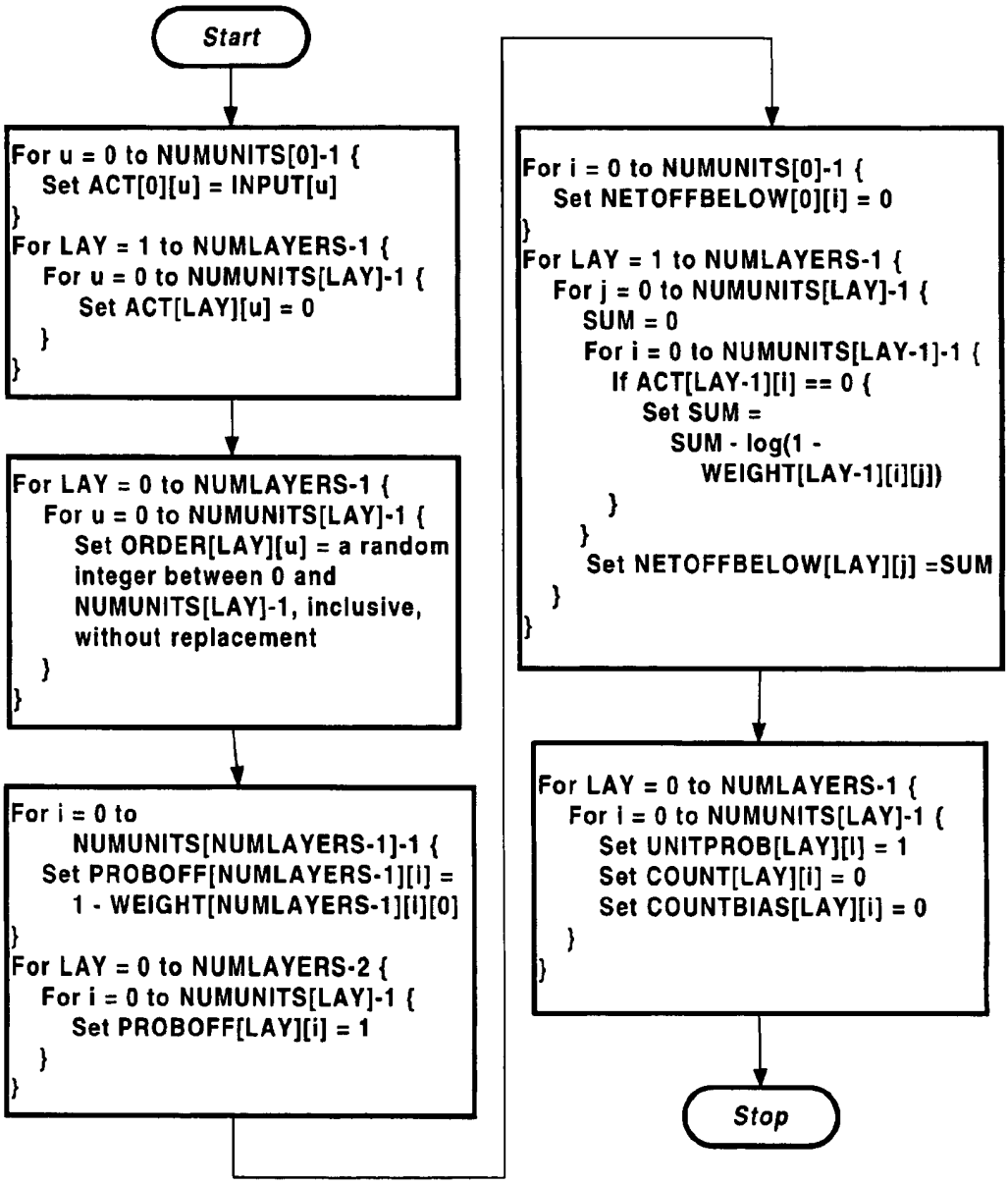

Start
For u = 0 to NUMUNITS[0]-1 {
Set ACT[0][u] = INPUT[u]
}
For LAY = 1 to NUMLAYERS-1 {
For u = 0 to NUMUNITS[LAY]-1 {
Set ACT[LAY][u] = 0
}
}
For LAY = 0 to NUMLAYERS-1 {
For u = 0 to NUMUNITS[LAY]-1 {
Set ORDER[LAY][u] = a random integer between 0 and NUMUNITS[LAY]-1, inclusive, without replacement
}
}
For i = 0 to NUMUNITS[NUMLAYERS-1]-1 {
Set PROBOFF[NUMLAYERS-1][i] = 1 - WEIGHT[NUMLAYERS-1][i][0]
}
For LAY = 0 to NUMLAYERS-2 {
For i = 0 to NUMUNITS[LAY]-1 {
Set PROBOFF[LAY][i] = 1
}
}
For i = 0 to NUMUNITS[0]-1 {
Set NETOFFBELOW[0][i] = 0
}
For LAY = 1 to NUMLAYERS-1 {
For j = 0 to NUMUNITS[LAY]-1 {
SUM = 0
For i = 0 to NUMUNITS[LAY-1]-1 {
If ACT[LAY-1][i] == 0 {
Set SUM = SUM - log(1 - WEIGHT[LAY-1][i][j])
}
}
Set NETOFFBELOW[LAY][j] =SUM
}
}
For LAY = 0 to NUMLAYERS-1 {
For i = 0 to NUMUNITS[LAY]-1 {
Set UNITPROB[LAY][i] = 1
Set COUNT[LAY][i] = 0
Set COUNTBIAS[LAY][i] = 0
}
}
Stop

Fig. 5

```
For LAY = 0 to NUMLAYERS-1 {
  For u = 0 to NUMUNITS[LAY]-1 {

    Set i = ORDER[LAY][u]

    If value of ACT[LAY][i] is not clamped {
      Select a new value for ACT[LAY][i]
      using Gibbs Sampling (see Fig. 7)
    }

    If value of ACT[LAY][i] changed {
      If LAY > 0 {
        For k = 0 to NUMUNITS[LAY-1]-1 {
          If ACT[LAY][i] == 1 {
            Set PROBOFF[LAY-1][k] =
              PROBOFF[LAY-1][k] * (1 - WEIGHT[LAY-1][k][i])
          }
          Else {
            Set PROBOFF[LAY-1][k] =
              PROBOFF[LAY-1][k] / (1 - WEIGHT[LAY-1][k][i])
          }
          If PROBOFF[LAY-1][k] > 1 {
            Set PROBOFF[LAY-1][k] = 1
          }
        }
      }
      If LAY < NUMLAYERS-1 {
        For j = 0 to NUMUNITS[LAY+1]-1 {
          If ACT[LAY][i] == 1 {
            Set NETOFFBELOW[LAY+1][j] =
              NETOFFBELOW[LAY+1][j] + log(1 - WEIGHT[LAY][i][j])
          }
          Else {
            Set NETOFFBELOW[LAY+1][j] =
              NETOFFBELOW[LAY+1][j] - log(1 - WEIGHT[LAY][i][j])
          }
        }
      }
    }

  }
}
```

Fig. 6

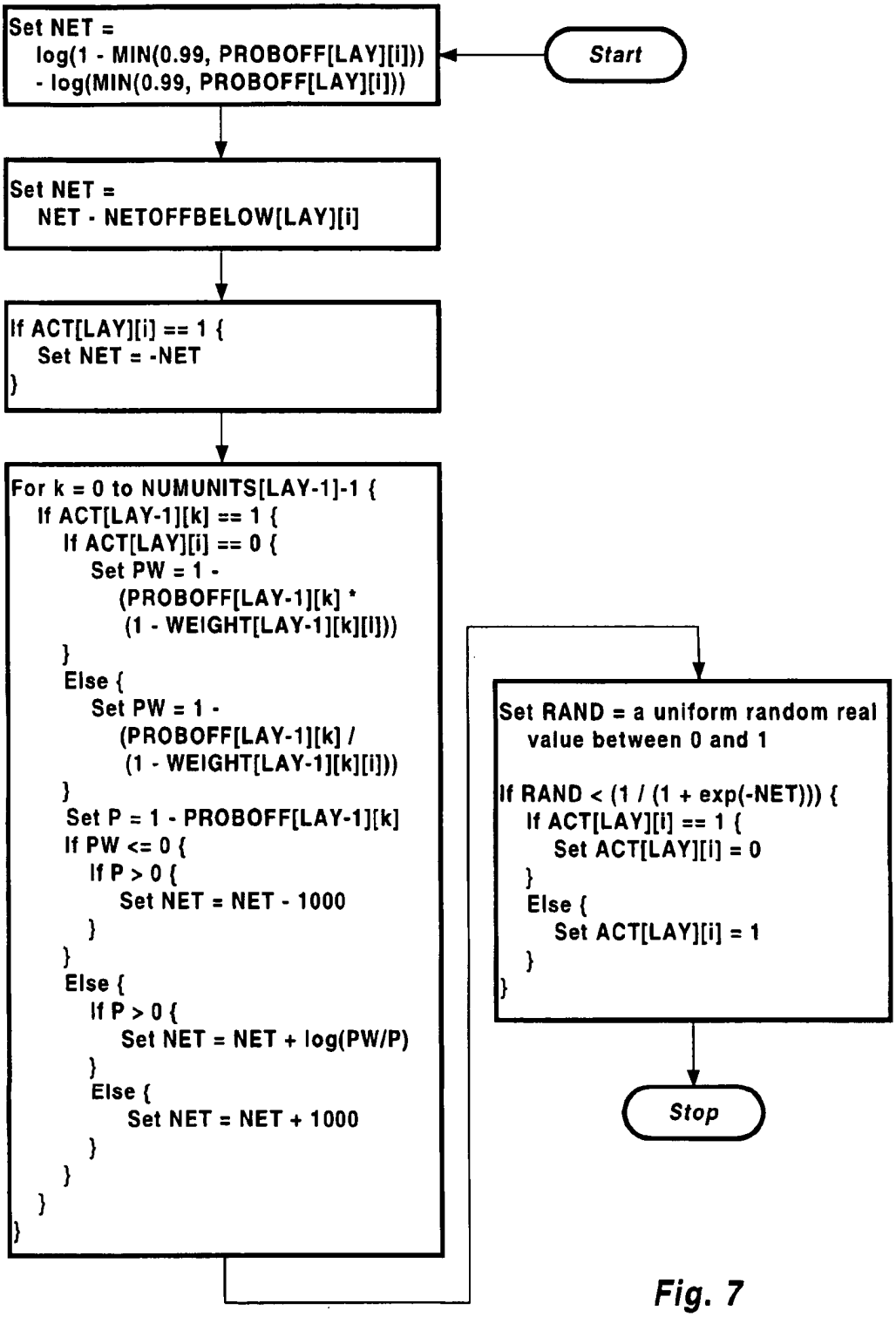

Start
Set NET =
log(1 - MIN(0.99, PROBOFF[LAY][i]))
- log(MIN(0.99, PROBOFF[LAY][i]))
Set NET =
NET - NETOFFBELOW[LAY][i]
If ACT[LAY][i] == 1 {
Set NET = -NET
}
For k = 0 to NUMUNITS[LAY-1]-1 {
If ACT[LAY-1][k] == 1 {
If ACT[LAY][i] == 0 {
Set PW = 1 -
(PROBOFF[LAY-1][k] *
(1 - WEIGHT[LAY-1][k][i]))
}
Else {
Set PW = 1 -
(PROBOFF[LAY-1][k] /
(1 - WEIGHT[LAY-1][k][i]))
}
Set P = 1 - PROBOFF[LAY-1][k]
If PW <= 0 {
If P > 0 {
Set NET = NET - 1000
}
}
Else {
If P > 0 {
Set NET = NET + log(PW/P)
}
Else {
Set NET = NET + 1000
}
}
}
}
Set RAND = a uniform random real
value between 0 and 1
If RAND < (1 / (1 + exp(-NET))) {
If ACT[LAY][i] == 1 {
Set ACT[LAY][i] = 0
}
Else {
Set ACT[LAY][i] = 1
}
}
Stop

Fig. 7

```
For LAY = 0 to NUMLAYERS-1 {
   For i = 0 to NUMUNITS[LAY]-1 {

      If ACT[LAY][i] == 1 {
         Set UNITPROB[LAY][i] =
            UNITPROB[LAY][i] * (1 - PROBOFF[LAY][i])
      }
      Else {
         Set UNITPROB[LAY][i] =
            UNITPROB[LAY][i] * PROBOFF[LAY][i]
      }

   }
}
```

Fig. 8

```
Set LRATE = 1

For LAY = 1 to NUMLAYERS-1 {
  For i = 0 to NUMUNITS[LAY]-1 {

    Set COUNTBIAS[LAY][i] = COUNTBIAS[LAY][i] + 0.05

    If ACT[LAY][i] == 1 {
      Set COUNT[LAY][i] = COUNT[LAY][i] + 0.05
      For k = 0 to NUMUNITS[LAY-1]-1 {
        Set D = -1
        If ACT[LAY-1][k] == 1 {
          Set D = D + (1 / (1 - PROBOFF[LAY-1][k]))
        }
        Set WPRE = WEIGHT[LAY-1][k][i]
        Set LPRE = -log(1 - WEIGHT[LAY-1][k][i])
        Set WEIGHT[LAY-1][k][i] = WEIGHT[LAY-1][k][i] +
          (D * LRATE * WEIGHT[LAY-1][k][i] / COUNT[LAY][i])
        If WEIGHT[LAY-1][k][i] > 0.99 { Set WEIGHT[LAY-1][k][i] = 0.99 }
        If WEIGHT[LAY-1][k][i] < 0.01 { Set WEIGHT[LAY-1][k][i] = 0.01 }
        Set PROBOFF[LAY-1][k] = PROBOFF[LAY-1][k] / (1 - WPRE)
        Set PROBOFF[LAY-1][k] = PROBOFF[LAY-1][k] *
          (1 - WEIGHT[LAY-1][k][i])
        If (PROBOFF[LAY-1][k] > 1 { Set PROBOFF[LAY-1][k] = 1 }
        If ACT[LAY-1][k] == 0 {
          Set NETOFFBELOW[LAY][i] = NETOFFBELOW[LAY][i] +
            (-log(1 - WEIGHT[LAY-1][k][i]) - LPRE)
        }
      }
    }

    If last cycle and LAY == NUMLAYERS-1 {
      Set D = -1
      If ACT[LAY][i] == 1 {
        Set D = D + (1 / (1 - PROBOFF[LAY][i]))
      }
      Set WEIGHT[LAY][i][0] = WEIGHT[LAY][i][0] +
        (D * LRATE * WEIGHT[LAY][i][0] / COUNTBIAS[LAY][i])
      If WEIGHT[LAY][i][0] < 0.01 { Set WEIGHT[LAY][i][0] = 0.01 }
      If WEIGHT[LAY][i][0] > 0.25 { Set WEIGHT[LAY][i][0] = 0.25 }
      Set PROBOFF[LAY][i] = 1 - WEIGHT[LAY][i][0]
    }
  }
}
```

Fig. 9

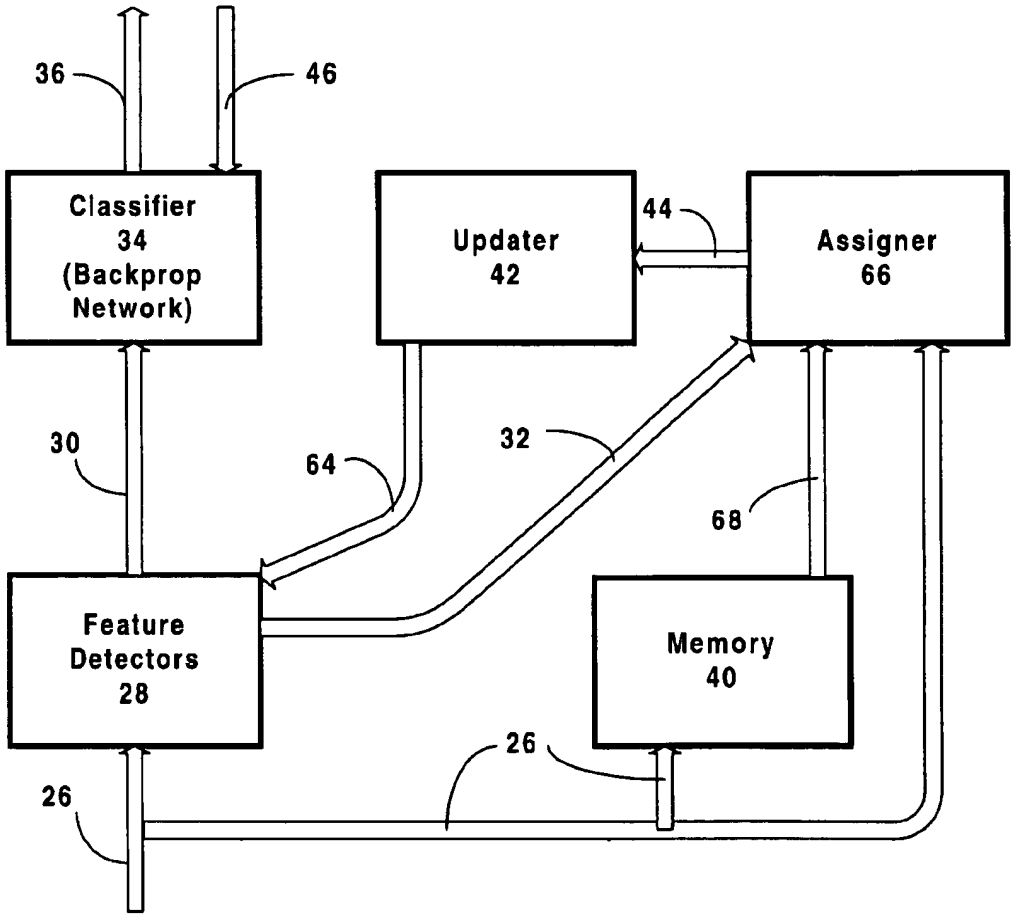

36
46
Classifier
34
(Backprop
Network)
Updater
42
44
Assigner
66
30
64
32
68
Feature
Detectors
28
Memory
40
26
26

Fig. 10

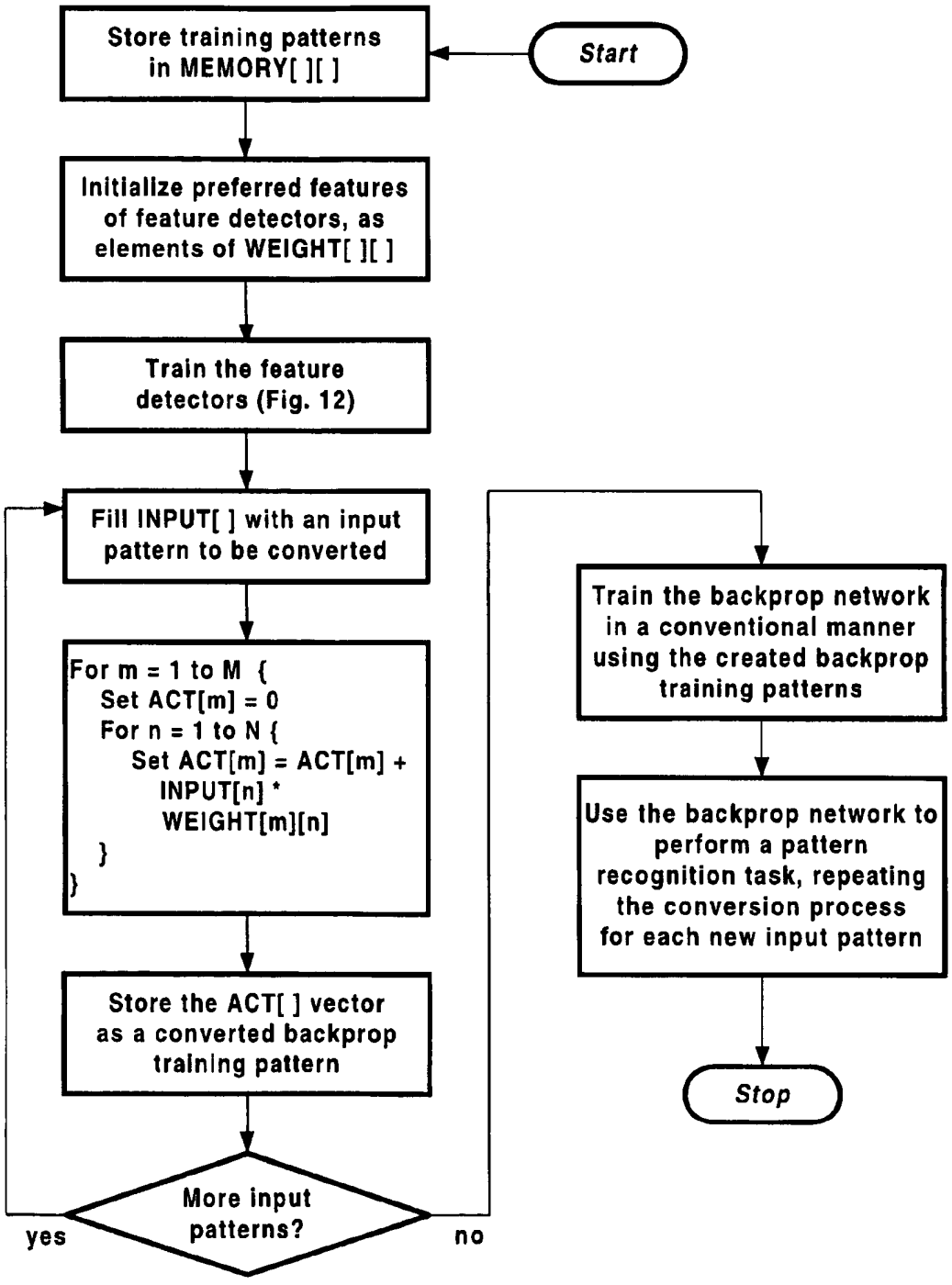

Start
Store training patterns in MEMORY[ ][ ]
Initialize preferred features of feature detectors, as elements of WEIGHT[ ][ ]
Train the feature detectors (Fig. 12)
Fill INPUT[ ] with an input pattern to be converted
For m = 1 to M {
Set ACT[m] = 0
For n = 1 to N {
Set ACT[m] = ACT[m] + INPUT[n] * WEIGHT[m][n]
}
}
Store the ACT[ ] vector as a converted backprop training pattern
More input patterns?
yes
no
Train the backprop network in a conventional manner using the created backprop training patterns
Use the backprop network to perform a pattern recognition task, repeating the conversion process for each new input pattern
Stop

Fig. 11

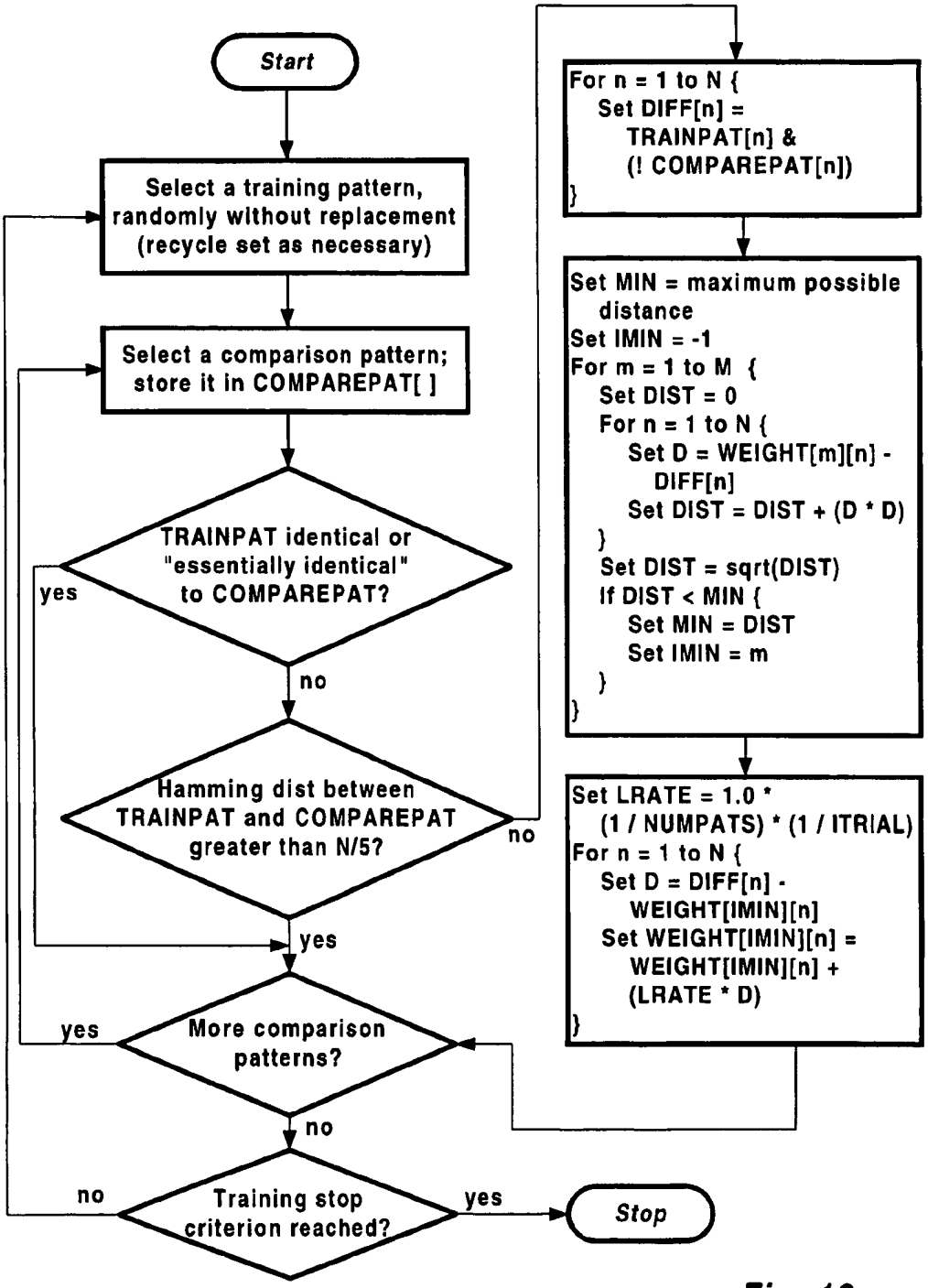

Start
Select a training pattern, randomly without replacement (recycle set as necessary)
Select a comparison pattern; store it in COMPAREPAT[ ]
TRAINPAT identical or "essentially identical" to COMPAREPAT?
yes
no
Hamming dist between TRAINPAT and COMPAREPAT greater than N/5?
no
yes
More comparison patterns?
yes
no
Training stop criterion reached?
no
yes
Stop
For n = 1 to N {
Set DIFF[n] =
TRAINPAT[n] &
(! COMPAREPAT[n])
}
Set MIN = maximum possible distance
Set IMIN = -1
For m = 1 to M {
Set DIST = 0
For n = 1 to N {
Set D = WEIGHT[m][n] - DIFF[n]
Set DIST = DIST + (D * D)
}
Set DIST = sqrt(DIST)
If DIST < MIN {
Set MIN = DIST
Set IMIN = m
}
}
Set LRATE = 1.0 * (1 / NUMPATS) * (1 / ITRIAL)
For n = 1 to N {
Set D = DIFF[n] - WEIGHT[IMIN][n]
Set WEIGHT[IMIN][n] = WEIGHT[IMIN][n] + (LRATE * D)
}

Fig. 12

PATTERN RECOGNIZER WITH INDEPENDENT FEATURE LEARNING

BACKGROUND—FIELD OF INVENTION

This invention relates to pattern recognition methods and machines, specifically to an improved method and machine for training feature-based pattern recognizers.

BACKGROUND—DISCUSSION OF PRIOR ART

Pattern recognizers can be used in many ways, all of which involve automatically responding to some physical pattern in the world. For example, the physical pattern might be speech sound waves, in which case a pattern recognition device might be used to output the same utterance but in a different language. Or the physical pattern might be the locations of vehicles on a particular highway, and the pattern recognizer might then be used to control the traffic lights on that highway so as to minimize congestion.

Often it is desirable to apply a pattern recognizer to a task which is poorly understood, or even a task which changes over time. In such circumstances, an adaptive pattern recognizer, which learns the task based on a sequence of examples, can work much better than a "hard-wired" (non-adaptive) one. Also, like adaptivity, "feature based" recognition can be very advantageous, in general because it tends to be more noise-tolerant than other approaches (such as fixed template matching). Feature based recognition involves responding to a set of features, or characteristics, which are determined to exist within the pattern. For example, if the patterns were speech waveforms, the features detected might include "a 'k' sound", or "a high-amplitude frequency within the twenty-eighth time interval". In a recognizer which is adaptive as well as feature-based, the features may even be very complex and difficult to describe in human language.

The device of the present invention is both adaptive and feature-based. One of the most difficult problems in designing such pattern recognizers is determining the best set of features—or more precisely, determining how the recognizer should be trained so that it will learn the best set of features. Very often, once a good set of features is found, the recognition problem becomes trivial.

One approach to learning good features is the use of a neural network trained with the backpropagation method (Rumelhart, Hinton, & Williams, 1986, "Learning internal representations by error backpropagation", in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, MIT Press, Cambridge, Mass.). However, this approach (and many related gradient-based neural net methods) can be very slow to learn, especially with networks having many layers of neurons. It is also quite possible that it will not learn optimal, or even nearly optimal, features. This is because it is based on a hill-climbing type of learning which can get stuck in a "valley" very far from the globally optimal solution. The result might be features which work well on the training examples, but not on new examples (i.e., poor generalization of learning).

There have been many attempts to improve on the learning speed or the generalization ability of these neural network pattern recognizers, but typically such improvements either do not solve both of these problems at once, or do not result in significant improvement on a usefully wide range of tasks. Arguably, the solutions which work best, though, tend to be ones which impose constraints on the learning process; that is, based on some assumptions about the task at hand, they constrain the learning so that only

certain feature sets can be (or are likely to be) learned. Such constraints effectively reduce the amount of "feature space" which must be searched by the learning process, making the process both faster and less susceptible to getting stuck with bad features.

One example of constraint-based feature learning is that of Simard et al. (e.g. Hastie, Simard, & Sackinger, 1995, "Learning prototype models for tangent distance", in *Advances in Neural Information Processing Systems* 7, MIT Press, Cambridge, Mass.). Their neural network-type method is applied to character (e.g. handwriting) recognition. In effect, their approach is to force the network to automatically generalize anything it learns about a particular example character to all possible "transformed" versions of the character. Here, the transformations include stretching, shrinking, slanting, and the like. While this does significantly improve generalization (and learning speed, since a smaller set of examples is required), the solution is rather specific to things such as writing. It wouldn't directly apply to speech waveforms, for example. A further disadvantage of this solution is that it only applies at the input level, where the features are first input to the neural network. It doesn't help at the internal layers of a multilayer network, because these internal features are learned, and thus it is not clear how to apply spatial constraints (e.g. slant-independence) to them. Similarly, this method doesn't address the problem such neural networks have in scaling up to large numbers of features. This scaling problem (which creates prohibitively long training times) results from the exponentially increasing number of possible feature combinations, and must be solved at all levels of feature detection in order to be significantly reduced.

OBJECTS AND ADVANTAGES

Accordingly, my invention has several objects and advantages over the prior art pattern recognition methods. Like backpropagation and some other neural network training methods, my invention may be used for adaptive learning within a feature based pattern recognition machine. It improves upon these previous methods however, in that it provides a strong constraint on the learning, thus reducing the learning time and reducing the likelihood of poor features being learned. This constraint is based on an assumption that the ideal (or "true") features occur independently (are not "correlated") in the set of physical patterns. Ironically, this assumption has often been invoked in the prior art, but prior to my invention has not been used to its fullest extent.

My method makes much more extensive use of the independent features assumption, making it very powerful. Because this assumption is not limited to a particular class of pattern recognition tasks (e.g. only optical character recognition), my invention's advantages are likely to be obtained on a wide variety of tasks. Furthermore, the assumption is actually more powerful when more feature detectors are used. This allows for potentially improved scaling of the method to larger recognizers, which has long been a goal of the neural network research community. Still further, the independent features assumption can be applied at every layer of a hierarchical, multilayer recognition device. This gives my device even more ability to speed learning and improve generalization when compared to constraint-based procedures which apply only at the input layer.

A further object of my invention is that multiple similar recognition systems can be created by training one such

system and transferring the resulting trained weights to another system.

Further objects and advantages of my invention will become apparent from a consideration of the drawings and ensuing description.

DESCRIPTION OF DRAWINGS

FIG. **1** is a block diagram of a pattern recognition system according to the present invention, showing subsystems common to both preferred embodiments.

FIG. **2** is a flow diagram of an overall procedure for operating the preferred embodiments.

FIG. **3** is block diagram showing the structure of the first preferred embodiment.

FIG. **4** is a flow diagram of an overall procedure for operating the first preferred embodiment.

FIG. **5** is a flow diagram depicting the initialization of parameters for the first preferred embodiment.

FIG. **6** is a flow diagram of a single cycle of Gibbs sampling as performed in the first preferred embodiment.

FIG. **7** is a flow diagram of a procedure for selecting a new unit activation within a cycle of the Gibbs sampling process of the first preferred embodiment.

FIG. **8** is a flow diagram of a procedure for updating a unit's contribution to the likelihood within a cycle of the Gibbs sampling process of the first preferred embodiment.

FIG. **9** is a flow diagram of a procedure for updating connection weights within a cycle of the Gibbs sampling process of the first preferred embodiment.

FIG. **10** is a block diagram showing the structure of the second preferred embodiment.

FIG. **11** is a flow diagram of an overall procedure for operating the second preferred embodiment.

FIG. **12** is a flow diagram of a procedure for training the feature detectors of the second preferred embodiment.

LIST OF REFERENCE NUMERALS

20 Environment and/or Further device
22 Physical pattern
24 Transducer
26 Input signal
28 Feature detectors
30 Feature activity signal
32 Feature description signal
34 Classifier
36 Output signal
38 Effector
40 Memory
42 Updater
44 Part mapping signal
46 Target signal
50 Class networks
64 Updating signal
66 Assigner
68 Retrieval signal
70 Action

SUMMARY

In accordance with the present invention a pattern recognition device comprises a sensory transducer, a group of feature detectors, a classifier, and an effector to automatically respond to physical patterns. The device further comprises the improvement wherein an assigner uses previous input patterns, as stored in a memory, to segment a current input pattern into parts corresponding to the feature detectors, and at least one feature detector is modified so as to increase its preference for its assigned part.

Theory of the Invention

I believe there are interesting theoretical reasons for the advantages of my invention. This section describes this theory as I currently understand it.

Machines and/or methods of pattern recognition which are feature-based can be very powerful. For example, consider a recognizer for printed characters which has detectors for such features as "horizontal line at the top", "vertical line on the right", etc. One reason this is a powerful approach is that a relatively small number of such feature detectors can cooperate to recognize a large number of possible characters. Indeed, the number of different recognizable characters increases exponentially with the number of features (although this exponential increase is both a blessing and a curse, as will be explained shortly). For example, using only 20 binary (on/off) features, over a million possible patterns can be recognized; with 1000 features, the number of possibilities is almost too hard to comprehend—and this is still a puny number compared to the number of neurons in a human brain!

Another advantage of feature based recognition is noise tolerance. Essentially, if "enough" of the features in a pattern are detected, recognition can be good even though feature detection is not. For example, a capital "A" could be recognized even if the "horizontal bar in the middle" was missed (perhaps due to a bad printer), simply because "A" is the only letter with the (detected) features, "right-leaning diagonal on the left", "left-leaning diagonal on the right", and "intersecting line segments at the top". There are many possible feature sets which might be used for character recognition, but these serve to illustrate the basic point of fault-tolerance.

As powerful as feature based recognition can be, it is still much more powerful when the features can be learned from examples, rather than being hardwired by a human designer. Such adaptivity underlies the recent research interest in neural networks, for example, which in their most typical form are just successive layers of (adaptive) feature detectors. Indeed, many would argue that human intelligence is so impressive partly because it is based on naturally occurring adaptive neural networks with billions of neurons, wherein each neuron can be viewed as a feature detector.

However, the power of adaptive feature based recognition has always come at a price. In particular, learning the features can be very slow, and can result in suboptimal features being learned. Moreover, this problem seems to get worse, the greater the number of feature detectors being trained. This is the "curse" aspect of the exponentially increasing number of feature combinations, as alluded to above.

I believe, though, that the curse is not as bad as the prior art literature would suggest. Indeed, I believe it can't be, or else human brains, with their billions of feature-detecting neurons, could not learn nearly as fast as they do. I further believe that my invention makes use of a principle which is also used by human brains. This principle is what I call "independent feature learning".

Most prior art recognizers (adaptive feature-based ones) perform training of the feature detectors in an essentially similar way: They first attempt to identify which (important) features are contained in the current input pattern. Then they modify all the feature detectors so as to make the entire

recognizer better at detecting that particular combination of features. Thus if a "T" was observed, the combination of features "top horizontal bar" and "middle vertical bar" might be reinforced. Importantly, this means that now whenever "top horizontal bar" is observed, "middle vertical bar" will be considered more likely as well, and vice versa. The recognizer has been taught that these two features are (to some extent) correlated in the set of possible input patterns.

The essence of my invention, on the other hand, is the assumption that features are not correlated; rather, they are assumed to be statistically independent of one another throughout the set of input patterns. An embodiment of my invention, upon observing the "T", might train one feature detector to better respond to "top horizontal bar", and another to increase its preference for "middle vertical bar", but without increasing the preference of any detector for the combination of these two features.

Why should this be a good training method? Because in the absence of evidence to the contrary, it is a good first guess that any given feature could occur within any other combination of features. A recognizer which has just "discovered" the "top horizontal bar" feature, for example, could also find this feature useful when it later encounters "E", "F", "I", "Z", "5", "7", and perhaps other symbols too. But if it had been trained that "top horizontal bar" implies "middle vertical bar" as well—as prior art recognizers typically learn when observing a "T"—it would later have to unlearn this information when it encountered the other symbols. In essence, my device is advantageous because it does not require such unlearning. Indeed, I believe that in a typical prior art training regime, the appropriate unlearning of spurious correlations often never occurs, because the amount of training patterns is just too small. Thus I believe my device can not only learn with many fewer pattern observations, but it can also learn better features based on those observations.

Moreover, I believe this advantage becomes more and more important as the number of feature detectors increases. This is because the number of feature combinations increases exponentially, so in some sense the amount of inappropriate correlation learning done by prior art recognizers also increases exponentially—and thus so does the amount of unlearning that must be done. This means my device has the potential for improved scaling to large numbers of feature detectors.

One might argue that the assumption of independently occurring features—which leads to my approach of training the feature detectors independently—is not necessarily appropriate in all situations. For example, what if (in some strange alphabet) the letter "T" was the only letter with a "top horizontal bar" or a "middle vertical bar"? Surely then it would be appropriate to train the recognizer that these features always occur together, wouldn't it? One answer to this is that in such a situation, the entire "T" would be a more appropriate feature to learn. More generally, features which are highly correlated with other features tend not to be very useful anyway; they tend to waste feature storage capacity in the recognizer. In any case, though, my invention does not prevent the learning of correlations among features. It simply makes independence of features the default assumption; this assumption can still be "overruled" by further learning.

Some prior art methods have invoked the principle of independence, in an attempt to encourage learning of "factorial", or "information preserving" internal representations. One example is that of Foldiak (1990, "Forming sparse representations by local anti-Hebbian learning", in Biological Cybernetics, 64:165–170), which incorporates "competitive" connections between the feature detectors to encourage them to learn different features. However, these prior art methods have not made use of the independence assumption nearly to the degree possible. For example, systems like Foldiak's which incorporate competitive connections can only discourage second-order dependencies (correlations), not higher-order ones, as does my device. Also, these systems often perform "batch" training, where weight changes are saved, and actually performed only after a set of pattern presentations. Unlike my device, such procedures do not allow the learning done on a current pattern to be immediately used for assisting future learning.

Furthermore, I believe it is typical of these prior art methods that an essentially distinct subsystem (e.g. separate connections, or an additional penalty term in the training cost function) is used to counteract the effects of an otherwise conventional training procedure. In such a method, I believe, the countermeasure will always lag behind the principal, error-reducing weight updates. My invention, in contrast, embeds the independence assumption in the principal (and only) weight update procedure; thus the dependencies need not be learned by a separate subsystem in order to be (later) removed.

Commonalities of the Preferred Embodiments

This section describes aspects of the invention which are common to both of my preferred embodiments, with reference to FIGS. **1** and **2**. Details of the preferred embodiments are given below.

Overview

FIG. **1** illustrates an overview of a pattern recognition device according to the invention. Direction of signal propagation is indicated by arrows. Propagation of multiple-element signals should be logically parallel for the most part, meaning most (and preferably all) of a signal's elements (each of which represents a scalar value) should be received by a corresponding processing stage before that stage performs its operations and outputs its results. For clarity, each signal communication line is given a reference label identical to that of the signal which uses that line.

Operation of the system (described further below) is regulated by a user, which may be simply a human user or may be or include a further device. This regulation includes the orientation of the system such that it interacts with an environment and/or further device **20** (referred to hereafter as simply the environment **20**). This orientation is such that a physical pattern **22** is communicated from the environment **20** to the system, in particular to a transducer **24**. The transducer **24** converts the physical pattern **22** into a representation of that physical pattern **22**, which conversion I take to include any "preprocessing" operations. This representation takes the form of an input signal **26**. I will often refer herein to the information represented by a given input signal **26** as an "input pattern".

A group of feature detectors **28** is connected to receive the input signal **26** from the transducer **24**. Each of the feature detectors **28** is configured to detect or "prefer" a certain feature vector when it occurs in the input signal **26**. Each feature detector **28**[*m*] outputs a corresponding feature activity signal element **30**[*m*] representative of the (scalar) degree to which it has detected its feature within the current input signal **26** (i.e., the degree to which the detector "fires", or finds a good "match" in the input signal **26**). In some embodiments this feature activity signal element **30**[*m*] may reflect the results of competition or other communication between the feature detectors **28**. Each feature detector

28[m] is also configured to output a multiple-element feature description signal element 32[m], which is representative of the feature the detector 28[m] prefers.

A classifier 34 is connected to receive the feature activity signal 30 from the feature detectors 28. The classifier 34 is also connected to receive a target signal 46 from the environment 20. The classifier 34 is configured, via training (using the target signal 46) and/or hardwiring, to produce an output signal 36 representative of an appropriate system response given the feature activity signal 30. For example, the output signal 36 may represent degrees of membership in various classes, such as the probability a handwritten character input is 'A', 'B', 'C', etc. It is important to note that the classifier 34, while named such to reflect its typical use, need not actually perform a classification per se. The important thing is that the output signal 36 it produces represents an appropriate system response, whether or not it also represents a class label.

An effector 38 is connected to receive the output signal 36, and is configured to take some action 70 in the world based on that signal 36. For example, if the system is being used to recognize handwritten characters, the effector 38 might store an ASCII representation of the most probable character into computer memory, perhaps to allow a user to send email using a device too small for a keyboard.

The feature detectors 28 are trained using a memory 40, an assigner 66, and an updater 42. The memory 40 is connected to receive the input signal 26. The memory 40 is capable of storing, possibly in an approximate ("lossy") way, a representation of a set of previous input patterns, which are called "comparison patterns" with respect to the current and future input signals 26.

The assigner 66 is connected to access the stored contents (comparison patterns) of the memory 40 via a retrieval signal 68. It is capable of using this storage to segment the current input pattern (as represented by the current input signal 26) into parts. Each part represents a vector which is judged by the assigner 66 to be a likely feature which is not only contained in the input signal 26, but is also likely to be useful for describing the collection of stored comparison patterns in the memory 40. Put differently, a part is a vector which is judged likely to be a true feature of the entire input domain, which includes past patterns, the current pattern, and (hopefully) future patterns as well.

The assigner 66 is connected to receive the feature description signal 32 and makes use of this signal 32 to create a part mapping signal 44, representative of a correspondence between feature detectors 28 and the parts. As described later, the memory 40 may also make use of the feature detectors 28 in storing input patterns. Furthermore, the assigner 66 may make use of the target signal 46 in creating the part mapping signal 44. The parts themselves may be explicitly represented (internally) by the assigner 66, or may only be represented implicitly in the part mapping signal 44.

The updater 42 is connected to receive the part mapping signal 44 from the assigner 66. It is configured to modify the feature detectors 28 based on this signal 44. In particular, the updater 42 can modify a feature detector 28[m] so as to increase the preference that the feature detector 28[m] has for the part corresponding to it. Put differently, the preferred feature of the feature detector 28[m] is moved toward, or made more similar to, the part assigned to it. The influence of the updater 42 is indicated in FIGS. 1, 3, and 10 by an updating signal 64. In some non-preferred embodiments, however, the feature updates might be made directly (e.g. via hardware), with no intervening updating signal 64 being required.

FIG. 2 illustrates an overview of the operation of a pattern recognition device according to the invention. Use of the device comprises a sequence of "trials", or physical pattern presentations. On each trial, either recognition is performed, or training is performed, or both are performed. In my preferred embodiments, recognition (if enabled) is performed before training (if enabled). However, many useful embodiments might exist wherein training is done before or simultaneous with recognition.

Both recognition and training require the physical pattern 22 to be observed, and a representative input signal 26 to be produced by the transducer 24. Other steps depend on whether recognition and/or training are enabled.

The schedule of enablement of training and recognition over trials is discussed below for each embodiment separately. One point should be made here, though. In my second preferred embodiment, the memory 40 is separate from the feature detectors 28, and input patterns are stored in the memory 40 before any training or recognition occurs. However, in my first preferred embodiment, the feature detectors 28 are actually used to implement the memory 40. In this case, storage of patterns in memory 40 is accomplished by the same procedure as training of the feature detectors 28. Thus, with respect to the first preferred embodiment, the step shown in FIG. 2 as "Store a set of patterns in memory" includes the setting of initial random preferred features, and possibly doing feature training on some number of patterns.

If recognition is enabled, the input signal 26 is communicated to the feature detectors 28, which evaluate the input against their preferred features and produce appropriate feature activity signal elements 30[1] through 30[M]. (In some embodiments an equivalent step is done as part of the training process, too.) The feature activity signal 30 (composed of the elements 30[1] through 30[M]) is used by the classifier 34 to produce an output signal 36. The output signal 36 is used by the effector 38 to take an appropriate action 70 within the system's environment 20.

If training is enabled, the input signal 26 is communicated to the memory 40, which may store the current input pattern information, and to the assigner 66. The assigner 66 uses the stored comparison pattern information, obtained from the memory 40 via the retrieval signal 68, to segment or parse the input signal 26 into parts. (In some embodiments the memory 40 may be implemented using the feature detectors 28 or their equivalent). The assigner 66 then uses the feature description signal 32 to assign the parts to corresponding feature detectors 28. The results of this assignment are communicated via a part mapping signal 44 to the updater 42. The assigner 66 may in some embodiments make use of the target signal 46 to perform the assignment. The updater 42 modifies the preferred features of the feature detectors 28. The modification is such that a feature detector 28[m] increases its preference for the part assigned to it.

After a significant number of training trials have occurred, the feature detectors 28 store valuable information about the input pattern domain, which may be used to bypass the training phase in a comparable pattern recognition device. Thus as shown in FIG. 2, the preferred features of one or more of the feature detectors 28 may be transferred (which includes being copied) to one or more comparable devices after some amount of training. A comparable device would be one having a transducer similar to the transducer 24, and one or more feature detectors similar to the feature detectors 28, so as to be capable of making appropriate use of the trained preferred features.

Implementation Details

Each of my preferred embodiments is implemented using a suitably programmed general-purpose digital computer. Generally speaking, signals and other representations will thus be implemented with storage space in the random access memory of the computer. Such an implementation is preferred in part because of the high availability and relatively low cost of such machines (as opposed to, for example, analog and/or non-electronic devices). Certain experimental manipulations may also be desirable, and these will typically be easiest to perform on a general-purpose machine via software. Furthermore, those skilled in the art of adaptive pattern recognition tend to be most familiar with software based implementations of pattern recognizers. Still further, such a system, once trained, can easily be used to create other systems to perform similar tasks, by copying trained weights and/or program code into other recognition systems.

In order to describe the computer program part of the preferred embodiments, variable names will be used to denote corresponding digital storage locations. These variables, along with the system parts which they implement, will be given with the preferred embodiment specifics below.

Pseudo-code Conventions

Some of the drawings make use of a "pseudo-code" which largely resembles the C programming language. One reason for this is to reduce the number of figures which must be used to represent a procedure. I believe this will make the overall methods depicted easier to understand by a typical pattern recognition programmer than if the methods were broken up into still more figures. In fact, the pseudo-code should be readily understandable by anyone skilled in C or a similar language. However, I will describe the least obvious conventions next.

Assignment to variables is indicated by a "Set var=value" statement. This is the equivalent of the C assignment operation "var=value".

A processing loop is indicated by a "For x=begin to end {loop-body}" statement. Here, loop-body is the code over which to loop, and x is an integer index variable whose value is typically referenced within the loop body. The loop is performed first with x equal to begin, and x is incremented by one before each successive iteration, until x is greater than end, at which point no further iterations occur.

Conditional code execution is implemented with an "If boolvar {conditional-code}" statement. Here, the conditional-code statements are executed if and only if the expression represented by boolvar evaluates to TRUE (nonzero). Sometimes I use an English language expression for boolvar, where the evaluation method is apparent. Also, a corresponding "Else { }" clause may be used with an "If" statement, as in C.

An array is often indicated by notation such as arrayvar[ ], or arrayvar[ ][0]. Such arrays represent vectors, as they have exactly one dimension with no specified index. Similarly, arrayvar[ ][ ] would indicate an entire two-dimensional array, and arrayvar[2][3] would indicate just a single element of a two-dimensional array. Also, array index brackets will be dropped for clarity when the context makes the meaning clear.

The operator "log" indicates a natural (base e) logarithm operation. The operator "exp" indicates a base e exponentiation operation. MIN(x, y) returns the minimum value of x and y.

Transducer

At the front end of the system is the transducer **24**, which senses a physical pattern **22** and produces an input signal **26** representative of it. The physical pattern **22** may be virtually any object or event, or conglomeration of objects and/or events, that is observable. Similarly the transducer **24** may be anything capable of detecting such an observable. It may include, for example, photodetector cells, a microphone, a camera, sonar detectors, heat sensors, a real-time stock quote device, a global position device implanted in a blind person's cane, and so on. It may detect electronically stored patterns, for example a stored hypertext document on a remote network server. The transducer **24** may also include one or more humans, as for example when survey results are observed. Those skilled in the art of adaptive pattern recognition will readily find many diverse physical pattern domains to which the present invention may be applied, as there are a great number of known methods and devices for sensing an extremely wide variety of patterns in the world.

The transducer **24** is also assumed to handle any necessary "preprocessing" of the physical pattern **22**. Preprocessing includes any known, hardwired transformations used to remove unwanted redundancy in the input, fill in missing values, and the like. These operations tend to be problem specific, and there are a great many possible ones. Some examples are: line extraction in character recognition; bandpass filtering of audio (e.g. speech) signals; and translation, rotation, and size normalization of images. It is important to note, though, that preprocessing is less important when using an adaptive feature-based device such as that of the present invention. While still useful, especially in well-understood domains, appropriate preprocessing can to some extent be "learned" by the adaptive part of the device. Because of this, in a worst-case scenario, wherein the system designer knows virtually nothing about the features contained in a physical pattern domain (and thus what preprocessing operations are appropriate), the present device can still be used without any preprocessing (i.e. with "raw" input data).

Those skilled in the art of adaptive feature-based pattern recognition will be familiar with methods for producing a sequence of input signals **26** and presenting these to a digital computer based recognizer in the form of a sequence of vector values. Thus herein it is simply assumed that the input signal **26** is available as the variable INPUT[ ]. Note that transduction (including preprocessing) may be done off-line; that is, recognition and/or learning may be performed on an input signal **26** obtained from stored data, as long as transduction occurred at some time to produce the stored data.

The variable INPUT[ ] is assumed to be binary with 0/1 values. If necessary, analog information may be converted to binary using the Albus Method (BYTE magazine, July 1979, p. 61, James Albus) or another such known method. I believe there are straightforward extensions of my preferred embodiments which will work on analog inputs, but since I have not tested these, binary representations are preferred.

Effector

The last stage of the recognition process is handled by the effector **38**. The effector **38** takes the output signal **36**, in the form of the vector computer variable OUTPUT[ ], and produces an action **70** in the system's environment **20** which is (after learning) as appropriate as possible for the current input signal **26**. As with transduction, this stage is well known in the prior art and thus will not be detailed here. Examples of effectors would be gears on a robot, traffic lights, a speaker, or a digital storage device. Combinations of different types of effectors might also be used. One use of a digital storage type effector would be to store an output signal **36** for future use. Such storage might, for example,

permit the invention to be used in implementing a database (perhaps of hypertext documents), wherein future queries would access the digitally stored outputs. In such an embodiment the effector **38** might, for example, store a copy of the input signal **26** along with an estimated class label obtained from the classifier **34** via the output signal **36**.

Notes on Experimentation

A certain amount of experimentation is inherent in the optimal use of adaptive pattern recognition, just because the pattern domain is never completely understood (or else an adaptive system would not be required in the first place). Thus adaptive pattern recognizers are best viewed as tools for solving a problem, rather than the solution itself. However, with reasonable experimental techniques, the performance gap between a perfectly optimized recognizer and a practically optimized one can be made much smaller. Furthermore, even a minimally optimized recognizer architecture, once trained, can often outperform any existing solutions, making it extremely valuable despite being "non-optimal".

In general, the experimental techniques appropriate for my preferred embodiments are the same as those known by those skilled in the art of adaptive pattern recognition. I will point out herein where any special considerations should be made with respect to my preferred embodiments. *The Handbook of Brain Theory and Neural Networks* (Arbib, ed., MIT Press, Cambridge, Mass.) is a very comprehensive reference for techniques related to adaptive pattern recognition, and also contains numerous references to related prior art reference material. The sections referring to backpropagation and unsupervised learning will be especially relevant, and will point to other relevant material. References such as these should be used to learn about appropriate experimental techniques, if not already known.

Preferred Embodiment 1

Architecture Figure and Flow Diagram

My first preferred embodiment is described with reference to FIGS. **3** through **9**. FIG. **3** illustrates the structure of the first preferred embodiment in more detail than in FIG. **1**. The environment **20**, transducer **24**, and effector **38** are left out of FIG. **3** for clarity. FIG. **4** provides a flow chart illustrating an outline of the software implementation in more detail than in FIG. **2**, and FIGS. **5** through **9** provide more detailed flow charts of the steps involved.

Theory

My first preferred embodiment makes use of a so-called "noisy-OR" neural network architecture. Radford M. Neal provides a good description of the theory of such networks, and provides further references ("Connectionist learning of belief networks", *Artificial Intelligence* 56, 1992, pp. 71–113). These references should be used to provide any required background beyond that provided herein, except with respect to the learning procedure. My learning procedure is different from the one described by Neal. Another description of noisy-OR networks is provided by Jaakkola and Jordan ("Computing upper and lower bounds on likelihood in intractable networks", in *Proceedings of the Twelfth Conference on Uncertainty in AI*).

A noisy-OR network uses binary (0/1 preferably) units, or neurons, which are activated according to a so-called noisy OR function. According to this method, there is a quantity p[i][j] for each pair of units i and j, representing the probability that unit j "firing" (having value 1) will cause unit i to fire as well. (Neal works instead with the values q[i][j], where q[i][j]=1−p[i][j].) There is also potentially a "bias" value for each unit, which is essentially a connection from a hypothetical unit which is always on (firing). My preferred embodiment uses bias weights only for the highest-level units, though.

In my preferred embodiment the units are arranged in layers, with the bottom layer representing the input pattern (i.e., corresponding to the input signal **26**). The goal in such a network is to learn an internal model of the pattern domain. This is also referred to as "unsupervised" learning. The internal model can also be viewed as a "pattern generator", and represents a probability distribution over the input pattern space. Ideally, a trained noisy-OR network could be used to randomly generate patterns whose distribution would very closely match the distribution of the training patterns. Because this type of network is most naturally viewed as a pattern generator, the connections will be said to feed from top to bottom in the network. However, in practice data flows both ways along the connections.

Recognition could occur in at least two basic ways in such a network. First, the network could be given a partial input pattern and its internal model used to fill in the missing input values (assuming during training there were input patterns without these values missing). The missing values could represent class labels, in which case the network could be used for classification. Note that classification learning is often viewed as "supervised" learning, but nevertheless a so-called unsupervised procedure can still be made to perform a similar task.

A second way of doing recognition with such a network—which is my preferred way—is to use a separate class network **50**[*c*] to model each class, as shown in FIG. **3**. At recognition time, the input pattern is presented to each class network **50**[*c*], and each is used to produce a likelihood value representing the probability that that network would generate the input pattern. These likelihood values are computed by the classifier **34**, which receives from the class networks **50** the feature activity signal **30** as well as other information needed to compute the likelihoods, such as the network weight values and activities of non-hidden units. The classifier **34** combines these likelihood values (via the well known Bayes Rule) with the prior class probabilities, to obtain the (relative) posterior class probability information. From this information it computes the index of the most probable class, which it communicates via the output signal **36**. Note that in this embodiment, any hidden unit in any network can be viewed as one of the feature detectors **28**.

The separate-networks approach has a drawback, which is that feature detectors cannot be shared by different classes. However, it circumvents a problem with the missing-value approach, which is how to force the network to learn features relevant for the classification task.

During recognition, the input signal **26** is presented to each class network **50**[*c*] in turn, and class likelihood values are computed as described, by the classifier **34**. During learning, however, the input signal **26** is only presented to the class network **50**[cTarget] corresponding to the (known) target class of the input pattern. Likewise, only the target network is trained on the current pattern. Since all the class networks **50** operate similarly, the network index will be suppressed herein for clarity where it is immaterial.

Both recognition and learning require an inference process whereby the network **50**[*c*] fits its internal model to the current input data (as represented by the current input signal **26**). Typical prior art implementations of noisy-OR networks use some sort of iterative inference process, in which multiple "activation cycles", or updates of unit activations, are performed. My preferred inference process is also of this type. Two results of this inference process are especially important. First, a likelihood value is produced which

(ideally) represents the probability that the network would produce the current input. This likelihood value is computed by the classifier **34** to facilitate classification. Second, the inference process produces a mapping from feature detectors **28** to input parts. In this case, each non-input unit is a feature detector **28**[*m*], and the "input part" corresponding to a unit is that part of the activation pattern in the layer below which the unit is judged to be "responsible" for producing (keeping in mind that we are viewing the network as a pattern generator).

My preferred inference process is Gibbs sampling, a technique known in the prior art. It is a statistically based process, involving successive random sampling of the units' activity states (on or off). Each unit is visited in turn, and its activity is selected from a distribution conditional on the current states of all the other units in the network. If this process is run "long enough", the distribution of network states will come to mirror their likelihoods, i.e. their respective probabilities, given the network's input data. An average over several such states can thus provide an estimate of the overall likelihood of the network model.

One of the virtues of this embodiment is that the inference process is iterative, and incorporates feedback. This effectively allows upper layers to influence how lower layer units are activated, when multiple hidden layers are used. Such top-down influences can lead to more flexible and accurate recognition overall. However, this extra power comes at a price: more processing time is required, relative to methods which are strictly feedforward. This extra processing can be minimized, though, by stopping the iterations when the changes they produce become less than some criterion.

Gibbs sampling is also used for learning purposes. For each state of the network, a responsibility value r[i][j] can be computed for each pair of units i and j, representing the responsibility that unit j had for causing unit i to fire. Note that r[i][j] is not the same as the value p[i][j] mentioned above. The p[i][j] is a hypothetical probability: the probability that unit i would fire if unit j were to fire. The value r[i][j], on the other hand, represents the effect unit j actually had on unit i, given a particular instantiated network state.

The array of responsibility values for two connected layers of units constitutes a segmentation; it indicates which "parts" of the lower layer activities "go with" which units in the upper layer. Each unit in the upper layer is judged responsible for a certain part of the activity in the lower layer (unless it is inactive, in which case it is not responsible for anything). Viewing the upper layer units as feature generators, the responsibilities indicate what features were generated to create the lower layer activities, and which units generated which features. The units can also be viewed as feature detectors of course, and a unit's preference for a given feature is directly related to its probability of generating that feature. Note also that units can share responsibility for a given "on" unit in the layer below. This will be referred to as "soft segmentation", as opposed to "hard segmentation" wherein only one unit is allowed to be responsible for an "on" unit.

Learning occurs by moving the p[ ][j] values for unit j toward the corresponding r[ ][j] values for unit j. Put differently, we can view the vector of unit j's responsibilities as the "part" of the input it is responsible for, and its vector of outgoing weights (the p[ ][j] values) as its preferred feature. In those terms then, the learning procedure is to make unit j's preferred feature directly more similar to the input part to which it was assigned. The details of the method are given below.

Notice that in this embodiment, the networks **50** not only contain the feature detectors **28** (that is, the "hidden units"—possibly multiple layers of them), the networks **50** also are used to implement the memory **40**. This is a beneficial outcome of using this type of unsupervised network: because the noisy-OR network used is designed to model its input environment, it is essentially a (lossy) memory of the past inputs from that environment on which it has been trained. Furthermore, if there are multiple layers of units, these will also be operable as multiple memories of the appropriate types: each layer of units, and its connections to the layer below it, implements a memory for patterns of activity over that lower layer. FIG. **3** indicates the use of the networks **50** in implementing the memory **40** by dataflow arrows in both directions between the two subsystems.

Implementation

As mentioned, the core of the first preferred embodiment is implemented in software on a general-purpose digital computer. Thus a conceptual mapping exists between the structural subsystem description of FIG. **3** and the concrete implementation description. This mapping is as follows.

The input signal **26** is implemented by the storage and subsequent retrieval from computer memory of a variable INPUT[ ] (note that the term "computer memory" should not be confused with the "memory **40**", although the former is used in implementing the latter, of course). The feature detectors **28** include all the hidden units of the networks **50**. The preferred features of the feature detectors **28** for a given network are stored in computer memory as an array variable WEIGHT[ ][ ]. The feature description signal **32** is implemented by storage and retrieval of appropriate elements of the WEIGHT array from computer memory. The feature activity signal **30** is implemented with the storage and retrieval of an array ACT[ ]. Implementation of the feature detectors **28** includes program code which computes the elements of ACT[ ], using Gibbs sampling. The classifier **34** is implemented by program code which computes a value for the variable OUTPUT; this includes code for computing individual network likelihoods, which are combined to produce OUTPUT. Storage and retrieval of OUTPUT implements the output signal **36**.

The memory **40** includes the code which performs Gibbs sampling to elicit likely network (ACT[ ][ ]) states. Implementation of the retrieval signal **68** includes storage and retrieval of PROBOFF variables for the network units. The memory **40** makes use of the feature activity signal **30** in computing the retrieval signal **68** (PROBOFF values). As will be elaborated below, the PROBOFF values for a given layer are a (particular kind of) combination of WEIGHT values from the feature detectors **28** in the layer above. Thus the memory **40** is a lossy memory, since the WEIGHT values cannot in general store an arbitrary number of patterns without some loss of information.

The assigner **66** is implemented with code that computes responsibilities for the network connections. This code is part of the weight updating code of FIG. **9**, which computes the responsibilities implicitly as described below. The part mapping signal **44** is implemented with temporary storage within the weight updating code of FIG. **9**. This code block also implements the updater **42**. The target signal **46** is implemented by a variable called TARGET, indicating the target class of the current physical pattern **22**.

Architecture and Parameter Selection

Certain aspects of the system architecture will be determined by the problem to be solved. The number of networks, C, will be equal to the number of classes to be recognized (e.g., in recognizing lower case English letters, C would be 26). The number of input units (those in the bottom layer) will normally be the same for each of the networks, and will

be determined by the input representation chosen. Recall that this representation is preferred to be 0/1 binary, but other than that, its composition is left to the designer. The creation of an appropriate input representation is a common task in the prior art pattern recognition literature.

Other aspects of the architecture will require educated guesses and possibly experimentation to achieve optimal performance. Again, this is characteristic of prior art devices as well. For example, the number of layers of units in each network could be varied. My experiments have only been performed with two- and three-layer networks (i.e., one and two layers of connections), but I have no reason to believe good results would not be obtained with more layers than this. Indeed, I believe more layers would be beneficial for many problems where the input domain contains large amounts of redundancy, such as raw images. The combination of this with the use of limited receptive fields (a technique now well known in the neural network literature) will probably be especially useful. As a general rule, the harder it is (for a person) to describe a class in terms of the input features, the more helpful it may be to have additional layers. However, a two-layer network is still preferred, with more layers added only if experimental resources allow. This also simplifies interpretation of the nomenclature herein: the activations of the single hidden layer are represented by the feature activity signal **30**—that is, the hidden units correspond to the feature detectors **28**—and the input units receive the elements of the input signal **26**. The description herein considers the number of layers a variable, though (NUMLAYERS), to make experimentation with additional layers straightforward.

The numbers of units in each (non-input) layer are also parameters of the embodiment, as with prior art neural networks. The first number tried should be the best guess at the number of independent features in the input domain ("input domain" here means the activities of the next lower layer of units). A typical method of experimentation is to start with a very small number of units in each hidden layer, and increase the number after each training run, as long as performance of a trained system (on a cross-validation data set) improves and experimentation time allows. It is also typical to find better overall performance when the number of units decreases from a given lower layer to an upper layer. This is because one job of the unsupervised network is to remove redundancy, and fewer units are required to represent the same information, as more redundancy is removed.

Because my preferred embodiment is strictly layered, with no connections which "skip" a layer, it is convenient to view the weight values for a given network as a three-dimensional matrix, where the first index corresponds to the layer number, the second to the receiving (lower layer) unit, and the third to the sending (upper layer) unit. Thus I will use the variable WEIGHT[LAY][i][j] to represent the weight value from unit j in layer LAY+1 to unit i in layer LAY (where the layers are indexed starting with 0 for the input layer).

Regulation of Trials (Pattern Presentations)

As shown in FIG. **4**, the overall course of learning and recognition is divided into trials, each trial involving the presentation of a single input pattern. Generally speaking, the user and the problem being addressed will determine on which trials learning and/or recognition will be enabled. Obviously, recognition will not be very good to the extent that learning has not occurred. Preferably though, classification error should be evaluated throughout learning on a separate cross-validation data set, and learning (which is done on a training data set, not the cross-validation set) terminated when the cross-validation error bottoms out and begins to increase. This technique is well known in the art. Other techniques may also be useful, however. For example, learning might be enabled throughout the lifetime of the device, perhaps to allow continual adaptation to a nonstationary environment (in such a case, it would be inappropriate to decrease the learning rate over time, though—see below).

Training patterns should be chosen independently and at random according to the input distribution to be learned. Note that the memory **40**, which is implemented using the feature detectors **28** in this embodiment, does not contain any patterns at first (although the initial random weights could be viewed as representing hypothetical stored patterns). After one or more training trials, however, it is considered to have approximately stored the trained patterns. These stored patterns thus become the comparison patterns for future training trials, and are used in finding likely features or parts within each future input signal **26**.

Before any learning, all weights (p[i][j] values) of all C networks should be initialized to small random values. These are stored in the array elements WEIGHT[LAY][i][j]. Preferably these should be uniform random in the range 0.02 to 0.04, but this could be an experimental parameter if resources allow such experimentation. During learning, the weights are preferably maintained in the range 0.01 to 0.99 (by resetting any weight that is beyond a limit back to that limit, after normal weight updating). The purpose of this is to prevent learning becoming "stuck" due to extremely low likelihoods, and to help prevent computed probabilities from exceeding machine-representable values. However, if experimentation is possible, and it is known or believed that important features in the input domain may occur with probabilities beyond this range, then these limits should be adjusted to compensate.

There are two variables for each unit i, COUNT[i] and COUNTBIAS[i], which are used to count training trials, as further explained below. These must be initialized to zero before any training trials are performed.

On each trial for which training is enabled, a variable TARGET is set to the index of the target class for the current physical pattern **22**. A test will be performed during the loop over networks (as shown in FIG. **4**) to determine if the current class, c, is equal to TARGET. If so, processing, including training, will continue on network c.

Regulation of Cycles (Gibbs Sampling Iterations)

As shown in FIG. **4**, an important part of each trial is a loop over "cycles". This is done separately for each enabled network (just the target class network, if only training is enabled, and all class networks if recognition is enabled). The process is the same for each network, though, so here it will be discussed in the context of a single network.

Each cycle includes a single Gibbs sampling of each unit in the network, as well as a computation of the likelihood of the activation state produced. Also, two variables for each unit, PROBOFF and NETOFFBELOW, which are described below, are updated on each cycle. If training mode is enabled, weights are also updated on each cycle.

The reader is referred to the prior art literature to review the theory behind Gibbs sampling. The basic idea, though, is that each unit's activation is periodically sampled according to its probability conditional on the current activations of all the other units. Eventually, using this procedure, each overall network state will occur approximately with a frequency according to its overall probability (given the instantiated network input). This is a useful property because it is often very difficult to directly compute the probability of a network state conditional on a given input.

Each time a unit's activation is sampled, two values must be computed: the probability of the entire network (given all current values of the other units) if the unit were to have activation 0, and the probability of the network if the unit had activation 1. The probability that Gibbs sampling will assign the unit an activation of 0 is just the first of these two values, divided by their sum. If a 0 activation isn't assigned, then the unit takes on a value of 1.

A theoretically equivalent way of doing the same thing is to compute the probability that a unit's activation should change. This is the method of my preferred embodiment. It turns out with the noisy-OR architecture that a given unit will only be affected by a certain group of other units. In particular, a unit's parents, and children, and "siblings" (other parents of its children) are the only ones which need be considered when sampling a unit's activation.

My preferred embodiment employs a strategy which is a further improvement over a straightforward implementation of Gibbs sampling. This strategy takes advantage of the fact that many computed values do not change from one cycle to the next, especially on later cycles. Thus, an "update" strategy is employed, whereby certain useful quantities are maintained from cycle to cycle, and updated whenever other changes in the network state require. These updates typically consume less processing time overall than would recomputing the values on each cycle.

Two main variables are maintained for each unit, designated herein as PROBOFF and NETOFFBELOW. The PROBOFF value for a unit represents the probability that a unit is off given its parents—a quantity which is very useful in computing a unit's probability conditional on the rest of the network. Since computation of PROBOFF involves a product over the unit's "on" parents, it only needs to be updated when the activation of a parent unit changes, or the connection weight from a parent changes. Furthermore, the update need only deal with the changed parent activation, rather than iterating over all parents again.

Whereas PROBOFF may be considered a contribution from a unit's parents to its activation probability, the NETOFFBELOW value for a unit stores the contribution from "off" child units. It only needs to be changed when a child's activation changes, or a connection weight to a child changes. This value is very useful because in computing a unit's probability, the contribution from all "off" child units is computed by simply summing NETOFFBELOW with contributions from other units. Furthermore, NETOFFBELOW is itself just a sum of (negative) logs of the appropriate 1−p[i][j] values; i.e., it requires no multiplications or divides to compute (a table lookup could be used to speed up the log operation, and/or each connection's −log(1−p[i][j]) value could simply be stored). The overall implication of this is that the contribution from "off" children is very fast to compute. Moreover, in many applications the ratio of "off" to "on" units may be considerably higher than 1.0. To the extent this is true, the overall time to perform Gibbs sampling can be much faster with my method.

Initialization Before Gibbs Cycles

Before any cycles occur, certain variables are initialized, as shown in FIG. **5**. The activations of the input layer, represented by the variables ACT[**0**][**0**] . . . ACT[**0**][N−1], are set to be equal to the input pattern as stored in the array INPUT[0] . . . INPUT[N−1]. These values will be "clamped" during Gibbs sampling, meaning they are unchanged (not sampled). However, in other embodiments, in which some input values are missing, it would be appropriate to allow the missing values to be "filled in" by Gibbs sampling, by treating the corresponding input units like the network's hidden (non-input) units. For each non-input layer, all the unit activations are initialized to zero.

Also before cycling, a random sampling order is chosen for each layer of the network. This is simply a (uniform) random permutation of the unit indices of the layer, which is used as the order to perform Gibbs sampling within the layer. These indices are stored as the variable ORDER [LAY][ ], where LAY is the layer index, and the other index ranges over the units in that layer. Note that it may work well to use a different random order on each cycle, but my tests have not done this, and it requires somewhat more time, so it is not the preferred method.

The PROBOFF values for the units are initialized as follows. For the top layer in the network (layer number NUMLAYERS-1, where the bottom layer is layer zero), the PROBOFF value for each unit is just 1.0 minus the bias weight for the unit; that is, PROBOFF[NUMLAYERS−1][i]=1.0−WEIGHT[NUMLAYERS−1][i][0]. (Note the bias unit is considered to be a fictitious unit 0 in layer NUMLAYERS.) For each non-top layer, all of its units' PROBOFF values are initialized to 1.0, reflecting the fact that all non-input units are initially off.

The NETOFFBELOW variable for each unit is initialized as follows. For units with no children (input units), NETOFFBELOW is set to zero (and is always zero). For each other unit j in a non-input layer LAY, NETOFFBELOW is the sum over inactive child units i of −log(1.0−WEIGHT [LAY−1][i][j]). (Note this is the natural logarithm, i.e. base e.) Note that since all but the input units start out with activations of zero, all but the units in layer 1 (parents of the input units) will compute this sum over all their children.

The UNITPROB variable for each unit is initialized to one, for all units. This variable will be used to accumulate the (product of the) units' individual contributions to the overall network likelihood, which is computed over the course of all the cycles performed.

Two other variables are used for each unit as well: COUNT and COUNTBIAS. These are used to keep track of the number of training cycles for which the unit has been active (for COUNT) or has been either active or inactive (for COUNTBIAS). These variables are used in training, to reduce the amount of feature modification which is done over time, thus helping the training process to converge.

Gibbs Sampling and Unit Variable Updating

FIG. **6** (along with FIG. **7**) illustrates the Gibbs sampling process for a single cycle, in more detail. The overall structure is two nested loops, the outer loop iterating over layers in the network (bottom to top), and the inner one iterating over units within each layer. The majority of the processing, as described next, occurs for a particular unit, the index of which is chosen from the permuted index list ORDER[LAY][u].

The process for sampling a value for the current unit's activation, stored in ACT[LAY][i], is illustrated in FIG. **7**. Note that, as illustrated in FIG. **6**, no sampling is done on units with clamped activations. In my preferred embodiment all the input units' activations are clamped, and no other units are clamped. However, a distinction between "clamped" and "input" units is made here to assist those skilled in the art who may wish to experiment with alternative embodiments in which this is not true.

The strategy behind the procedure in FIG. **7** is to compute the probability that the current unit should change its activation, based on all the other units' current activations. The variable NET is used to accumulate "evidence" for the necessity of an activation change. NET is used as input to a sigmoid function, which outputs the probability of change.

This probability value is compared to a random real value between 0 and 1 inclusive to determine if a change is actually made.

NET is initialized using the contribution from the current unit's parent units. Most of the computation which goes into this has already been done, in the ongoing updates of the PROBOFF value for the current unit. The contribution is, theoretically, the (natural) log of (1−PROBOFF), minus the log of PROBOFF. This assumes that the current activation is zero; if it is not, the contribution must be multiplied by −1.

Notice that an adjustment is made to the theoretical value, though. In particular, instead of using PROBOFF directly, we use the smaller of PROBOFF and a constant, 0.99. This is done for the same reasons that the weights are clipped: to prevent problems of machine representation of small numbers, and to keep the Gibbs sampling from getting "stuck" due to extreme probabilities. Once again, though, if there is a reason to believe that this value 0.99 is too restrictive given the problem at hand, then experimentation should be done with a less restrictive (larger) value.

The second contribution to NET comes from the child units which are "off". Again, this has essentially already been computed via our scheme of updating, this time in the variable NETOFFBELOW. In particular, NETOFFBELOW is subtracted from NET. This assumes again that the current unit is "off". If it is not, NETOFFBELOW should be added to NET; this is done by the following conditional, as shown in FIG. **7**.

The contribution from "on" child units cannot be easily computed from running variables, as can the other contributions; it must be recomputed each time by iterating over all the (on) child units. This is done next in FIG. **7**. For each "on" child unit, we must compute the probability of that child having its current value, under two scenarios: (1) the current unit's activation changes, and (2) it doesn't change. In fact, it is the log of the ratio of these two probabilities which is added to NET, for each "on" child. This basic procedure is somewhat complicated by some enclosing conditionals. The purpose of these conditionals is just to handle the abnormal cases wherein one or the other, or both, of the probabilities is zero.

As shown in FIG. **6**, once an activation has been selected for the current unit (ACT[LAY][i]), a check is done to see whether the activation changed (the previous value must have been stored, of course). If it has changed, then the running variables PROBOFF and NETOFFBELOW must be updated for all other units which might be affected.

The PROBOFF value for a unit keeps track of its probability of being off, given its parents. Thus any units in layer LAY−1 need to have their PROBOFF value updated (of course if LAY is the input layer, there will be no such units). For each child unit k, this is done by either multiplying or dividing PROBOFF[LAY−1][k] by the probability that unit i would not turn unit k on—that is, by the quantity 1−WEIGHT[LAY−1][k][i]. Whether a multiplication or a division is performed depends on whether unit i is now off or is now on. Notice that topmost layer units will never have their PROBOFF values changed during cycling, because their only parent is the (hypothetical) bias unit, which has a constant activation of 1.

The NETOFFBELOW value for a unit i keeps track of the contribution to its probability from its "off" children. Thus any units in layer LAY+1 need to have their NETOFFBELOW value updated, since unit i in layer LAY has now changed its activation (of course if LAY is the topmost layer, their will be no such units). For each parent unit j, this is done by either adding to or subtracting from the variable NETOFFBELOW[LAY+1][j], the quantity −log(1−WEIGHT[LAY][i][j]). Whether an addition or a subtraction is performed depends on whether unit i is now off or on.

After performing Gibbs sampling for all the units, another double loop is performed as shown in FIG. **8**. Each unit in the network is again visited in turn (random index selection is not necessary here), and the UNITPROB value for each unit is updated. UNITPROB is ultimately used to estimate the likelihood of the entire network model, based on the current input pattern. This likelihood is the product of the individual unit probabilities (the probability each has its current activation, given the input). Furthermore, this quantity should be computed over a reasonably large sample of Gibbs cycles. My preferred method is to compute it over all the cycles (which is 20, preferably). Thus on each cycle, each unit's UNITPROB is simply multiplied by its PROBOFF value, if it is off, or 1 minus its PROBOFF value, if it is on (the UNITPROBs were initialized with value 1), as shown in FIG. **8**.

In practice, it may be preferable to do the UNITPROB computation in the log-probability domain, however, since multiplying many probability values together can create numbers which are too small to represent on some computers. In this case UNITPROB would be a sum of logs (initialized to zero) and the update would be to add log (PROBOFF) if the unit is off, and add log(1−PROBOFF) if the unit is on. While this procedure may avoid representation problems, it will also require more computation unless a log lookup table is used.

To the extent experimentation is possible, it may also be useful to try computing UNITPROB values just over the later cycles, for example just the last half of the cycles. This could potentially produce a more accurate estimate of the true network likelihood, because the Gibbs sampling will have had more time to settle toward the true distribution. However, there is a tradeoff when the total number of cycles is limited (as it must be in practice), because reducing the number of cycles used to do the estimation also reduces the quality of the estimate. Experimentation is the only way to find an optimal tradeoff; however, I believe my method will produce good estimates in general.

Feature Modification

After Gibbs sampling of each unit, and updating of appropriate running variables, feature modification (learning) occurs for the cycle, as shown in FIG. **9**. Of course, this is assuming that training mode is enabled; if the system were in recognition-only mode, no feature modification would take place.

The first step shown in FIG. **9** is to set a learning rate variable, LRATE, to 1.0. Since LRATE gets multiplied by each potential weight change, using a value of 1.0 is equivalent to not using a learning rate at all. However, one is used here because certain modifications of the preferred embodiment might require one, so it is useful to illustrate how LRATE should be used in the more general case.

As with Gibbs sampling and the updating of UNITPROB values, learning is done within a nested double-loop, over layers and units within each layer. The units are visited in turn, rather than according to a random index order, in my preferred embodiment. However, if experimentation is possible, I advise trying a modified embodiment in which the units within a layer are visited in a different random order on each cycle. This is because PROBOFF values for the layer below are modified during training of a unit, and this affects future training of other units in the layer. Thus, in my embodiment there is a bias according to a unit's index. While I don't believe it would make a significant improve-

ment to remove this bias with random indexes, it is possible that it could for some recognition tasks.

As each unit is visited, 0.05 is first added to its COUNT-BIAS value. This variable keeps track of the number of trials of learning which the unit has "experienced" so far. The value is 0.05 because 20 cycles are used in my preferred embodiment, and 0.05=1/20. A similar variable, COUNT, keeps track of just the number of training trials for which the unit was active. COUNT is updated within the conditional described next.

The weights leaving a given unit i (those to the layer below) are only modified if unit i is active. If it is, its COUNT variable is updated as just mentioned, and then a loop is entered to iterate over child units of i.

For each child k of unit i, we can compute an associated "responsibility" value, representing the responsibility unit i had in causing k to be active. If k is not active, this responsibility is zero. Otherwise, the responsibility is determined by dividing WEIGHT[LAY-1][k][i] by the quantity 1-PROBOFF[LAY-1][k]. This is essentially the prior probability that unit i would turn k on (WEIGHT[LAY-1][k][i]), divided by the prior probability that k would be on given all its parents' current activations. Note that we say "prior" here, because these probabilities do not take into account whether or not k is actually on as a result of Gibbs sampling.

The array of responsibilities for all of unit i's children constitutes the "part" of the pattern of activity in the child layer which has been assigned to unit i. The goal of learning is to move unit i's preferred feature—namely, its vector of weights going to its children—toward its assigned part. Thus we can view the vector of i's responsibilities as the "target" toward which we want its weights to be modified.

The upshot of this, in terms of an actual procedure for each weight, is that WEIGHT[LAY-1][k][i] should be moved toward zero, if unit k is not active on this Gibbs cycle, and towards WEIGHT[LAY-1][k][i]/(1-PROBOFF [LAY-1][k]) otherwise. (Remember that no changes are made at all unless unit i is active.) This is what the procedure of FIG. 9 does, although it does not explicitly compute the responsibility (target) value. Furthermore, the actual amount of the change is determined by LRATE and the COUNT value for unit i.

I believe the procedure of reducing the effective learning rate (i.e., LRATE/COUNT) using COUNT is the best way to achieve a balance of fast learning and convergence toward a stable solution. However, there are two related situations where this would not be so appropriate, and thus these situations are not preferred applications of my preferred embodiment. The first situation is where input patterns for the recognition system are not chosen independently and at random. The second situation is where the patterns are chosen at random, but the distribution changes over time (is "nonstationary"). In either of these cases, there could be an unwanted "primacy effect" due to the fact that more training is done on earlier patterns than on later ones. Although I do not recommend applying my preferred embodiment to such cases, if it were to be attempted, I believe the most appropriate approach would be to use a constant LRATE of considerably less than 1.0, and to not divide by COUNT.

After a weight is updated, it is then clipped to lie in the range 0.01 to 0.99, as discussed previously. Also, the PROBOFF and NETOFFBELOW values which depend on the just-modified weight are updated as appropriate. Note that while this might seem to incur a lot of computation, since there are so many weights, and a multiply and a divide are required each time, the situation is not as bad as it first appears. This is because learning only takes place for weights from an active unit, and furthermore in many applications there will be fewer active units than inactive ones.

Once a unit's outgoing weights have been modified (or not, if it was not active), a test is made to decide if its bias weight should be modified. Only topmost units even use a bias in my preferred embodiment, so that is one condition of the test. Also, bias weights are only updated on the last of the (20) cycles. For the most part, updating a bias is the same as updating any other weight. There is no need to test if the parent unit is active, though, because the bias unit is always active, albeit in a hypothetical sense.

Another exception is that my preferred embodiment maintains bias weights in the range 0.01 to 0.25. This is because my experiments showed that allowing a bias to grow too large could allow it to "dominate" the others: it would grow large, and would thereby take responsibility for nearly all inputs, thus creating a vicious circle in which other units could never "win" any inputs. However, as with the range limitation of the other weights, if there is reason to believe that the top-level "true" features in the pattern domain can occur with probability greater than 0.25, experimentation with an appropriately increased maximum should be done insofar as possible.

When a bias is changed, the PROBOFF value for the unit must also be updated. Since only topmost units have biases, though, and they have no other incoming connections other than biases, this is a simple update procedure, as FIG. 9 indicates.

Exiting the Cycle Loop

As shown in FIG. 4, once a cycle of Gibbs sampling has been done, along with the corresponding updates of weights and other variables (such as PROBOFFs), a check is made to decide whether to exit the cycle loop. In my preferred embodiment, as mentioned above, the loop is exited after 20 cycles have been performed. Another possible embodiment, though, would be to exit the cycle loop as soon as the amount of changes to the unit activations has become small, according to some measure. For example, the loop might be quit after two complete cycles had failed to produce any activation changes, or after 5 cycles wherein less than 2 percent of the unit activations changed. Obviously there are an unlimited number of similar strategies.

Such an alternative embodiment would have the advantage that when one interpretation of the input (i.e., one set of network activations) is much more likely than the rest, very little cycling would be required. This could often be the case once extensive training has already been done. However, one would need to deal with the issue of how much training to do on each cycle, given that a different number of cycles would be done on different patterns (this could be especially tricky for bias weights). Also, some maximum number of cycles would still need to be set. These added complications are the main reasons that I do not prefer such an embodiment.

System Output Determination

Compute a Relative Probability for the Network

As mentioned above, the Gibbs cycling process (including weight and variable updating) is the same for each network in the recognition system. This is also true of computation of a network probability value, which takes place after the cycling, unless only training mode is active, in which case the probability value is unnecessary. The probability of network c is stored in the variable NETWORKPROB[c] once computed, which will be used in computing an output for the overall recognition system.

Computation of NETWORKPROB values is easy, given that we have already computed UNITPROB values for all

the units in the network. NETWORKPROB[c] is just the product over all layers LAY and units i in network c of UNITPROB[LAY][i]. (Of course, if one were using the modified method described above of using log probabilities for UNITPROB, then NETWORKPROB[c] would be a sum of the UNITPROB values instead.) The NETWORKPROB [c] variable represents the probability of the network c model (as embodied by its architecture and modifiable weights, and as estimated here by a sampling of probable activation states) AND the input signal **26** (again, viewing the network as a generator of input signals). The NETWORKPROB values can thus be compared to see which is more probable for this particular input signal **26**.

Setting OUTPUT

As shown in FIG. **4**, once all networks in the system have been processed—for recognition, training, or both, depending on the system mode—the network loop is exited. If only training mode is enabled, processing is now complete for this input signal **26**. However, if recognition mode is enabled, the system output must be determined.

The system output is stored as the variable OUTPUT[ ], and is simply the index of the network which is most probable, given the current input signal **26**. (Note that OUTPUT[ ] is a single-element array here.) This index is then used as appropriate by the effector **38**, as described previously. If, as is preferable, the classes in the pattern domain are equally probable a priori, OUTPUT is just the index of the network with the largest NETWORKPROB value.

It is often the case, though, that classes in the pattern domain have different prior probabilities. In this case, an estimate should be made of these probabilities ("priors"), and stored in the array CLASSPROB[ ]. Then, for each class c, NETWORKPROB[c] should be multiplied by CLASSPROB[c], with the result stored in NETWORKPROB[c] (unless logs were used for the UNITPROBs and NETWORKPROBs, in which case the log of the CLASSPROBs should be added to the corresponding NETWORKPROB values). The NETWORKPROB values can then be compared in the same way as if the prior probabilities were equal.

Once the appropriate action **70** is taken based on OUTPUT[ ], processing of the current input signal **26** is complete. The next step is to (potentially) select a new input signal **26**, and repeat the processing for a trial (see "Regulation of trials" section above).

Preferred Embodiment 2

Architecture Figure and Flow Diagram

My second preferred embodiment is described with reference to FIGS. **10** through **12**. FIG. **10** illustrates the structure of the second preferred embodiment in more detail than in FIG. **1**. FIG. **11** provides a flow chart illustrating an outline of the software implementation in more detail than in FIG. **2**, and FIG. **12** provides a more detailed flow chart of the steps involved in training the feature detectors **28**.

Theory

My second preferred embodiment is different in many respects from the first preferred embodiment, and thus indicates to some extent the range of useful embodiments made possible by the invention. It uses independent feature learning to create a data compression device, which is then used as the front end to a (well-known) backpropagation network.

One way to make an intelligent guess as to what features are contained in a pattern is to use the existing feature detectors to segment it; this is the technique used by my first embodiment. Another way, though, is to use actual previous patterns, stored in memory; this is the approach used by my second embodiment. The heuristic underlying this strategy is the following: A feature could be defined as that which distinguishes two similar, but nonidentical, patterns—i.e., the "difference" between them. Therefore, a reasonable way to find likely features in a pattern is to compare it to stored patterns which are similar but not identical, and to compute some sort of difference for each such comparison. Having done this, the likely features—which are "parts", in the terminology of this invention specification—can be used to train the existing feature detectors.

The overall approach of the second preferred embodiment, then, is the following. The memory **40** is used to store, in a lossless manner, a large number of "comparison" patterns from the input domain to be learned. Each new pattern which is input is compared to one or more comparison pattern, and a difference vector DIFF[ ] is generated for each comparison. The assigner **66** compares each difference vector (part) to the preferred feature of each feature detector **28**[*m*], as communicated by the feature description signal **32**[*m*]. The detector **28**[*m*] which best matches the difference vector "wins" that difference vector, and this information is communicated to the updater **42** via the part mapping signal **44**. The updater **42** moves the winning detector's preferred feature toward the difference vector which it won, by some amount.

After a sufficient amount of such training, the feature detectors **28** are used as an input layer to a backpropagation based neural network, which plays the role of the classifier **34**. This is done by making the feature activity signal **30** the input to the backprop network, and having each of the feature detectors **28** become active to the extent that it's preferred feature is found in a subsequent new input signal **26**. Conventional supervised training is then done on the backprop network using these pretrained feature detectors **28**. The result is a pattern recognition system which requires fewer resources due to the learned data compression input layer. Furthermore, since the trained feature detectors **28** represent valuable information about the pattern domain, their preferred features may be copied or otherwise transferred to a comparable recognition system in order to avoid training the comparable system.

Because the memory **40** and feature detectors **28** are separate in this embodiment, the segmentation into parts (likely features) is likely to be not as good overall as that of embodiments (such as my first preferred one) which tightly integrate these two subsystems. Also, data compression is a lossy procedure, and as with other unsupervised procedures, there is no inherent way of forcing the learning of features which are relevant for the classification task at hand. For these reasons, this embodiment especially should be used as a tool, and only when experimentation is possible—not as a "quick-fix" solution for a mission-critical task. Of course, this is true to some extent of all adaptive pattern recognizers, including my first preferred embodiment, to the extent that the pattern domain is not well-understood.

While this embodiment lacks a tight integration of feature detectors **28** and memory **40**, it is also somewhat more simple to implement than embodiments such as my first preferred one. Furthermore, it allows a very wide variety of backprop networks to be used as the classifier **34**, which makes it a very flexible and powerful pattern recognition tool.

Implementation

As mentioned, the core of the second preferred embodiment is implemented in software on a general-purpose digital computer. Thus a conceptual mapping exists between

the structural subsystem description of FIG. **10** and the concrete implementation description. This mapping is as follows.

The input signal **26** is implemented by the storage and subsequent retrieval from computer memory of a variable INPUT[ ] (note that the term "computer memory" should not be confused with the "memory **40**", although the former is used in implementing the latter, of course). The preferred features of the feature detectors **28** are stored in computer memory as an array variable WEIGHT[ ][ ]. The feature description signal **32** is implemented by storage and retrieval of appropriate elements of the WEIGHT array from computer memory. The feature activity signal **30** is implemented with the storage and retrieval of an array ACT[ ]. Implementation of the feature detectors **28** includes program code which computes the value of ACT[ ]. The classifier **34** is implemented by program code which provides the functionality of a (conventional) backpropagation network. This backprop program code computes a value for the variable OUTPUT. Storage and retrieval of OUTPUT implements the output signal **36**.

The memory **40** includes computer storage and program code for storing, in their entirety, a sequence of training patterns, each represented by a distinct input signal **26**. Implementation of the retrieval signal **68** includes storage and retrieval of COMPAREPAT values, each of which represents a "comparison" pattern, and is one of the training patterns. The assigner **66** implementation includes code that computes a difference between a current training pattern, TRAINPAT, and a current comparison pattern, COMPAREPAT. It also includes storage for a variable DIFF[ ], representing this difference. It further includes code for finding the feature detector **28**[IMIN] whose preferred feature best matches DIFF[ ]. The part mapping signal **44** is implemented by storage and retrieval of the variables DIFF[ ] and WEIGHT[IMIN][ ]. The updater **42** implementation includes code for modifying WEIGHT[IMIN][ ] in the direction of DIFF[ ].

Architecture and Parameter Selection

Certain aspects of the system architecture will be determined by the problem to be solved. The number of input units (those in the bottom layer) will be determined by the input representation chosen. Recall that this representation is preferred to be 0/1 binary, but other than that, its composition is left to the designer. The creation of an appropriate input representation is a common task in the prior art pattern recognition literature.

This embodiment only has one layer of independent feature learning, which includes the weights to the feature detectors **28** from the input units (whose activities are communicated by the input signal **26**); these weights embody the preferred features, and will be stored as the variable WEIGHT[ ][ ]. However, the backprop network architecture may have multiple layers of connections. The considerations here will be the same as those in the prior art for backprop nets, with the extra consideration that the inputs to the backprop net will come from a data compression layer. If anything, this may eliminate the need for one layer of a backprop net which would have been used without data compression; but preferably the backprop architecture should not be changed from what it would have been without data compression.

The number of feature detectors **28** to use in the unsupervised layer—which corresponds to the number of input units in the backprop part of the system—is a parameter which will require experimentation to achieve an optimal value. This is characteristic of prior art devices as well, when they have hidden units. Normally the number should be less than the number N of input units in the unsupervised network; otherwise, there is no data compression occurring. The first number tried should be the best guess at the number of independent features in the input domain. A typical method of experimentation is to start with a very small number of units in the layer, and increase the number after each training run, as long as performance of a trained system (on a cross-validation data set) improves and experimentation time allows.

The backprop layers may be constructed in accordance with any compatible feedforward backprop network to be found in the prior art (to be compatible, the backprop architecture must allow M real-valued inputs, of magnitude possibly greater than one). The inputs to the backprop network will be the transformed input signals **26**, where the transformation uses the feature detectors **28**; that is, the input to the backprop net will be the set of feature activity signal elements **30**[*m*]. Note that for a typical backprop network, a target signal **46** will be required for each input signal **26**, representing the desired output signal **36** for that input signal **26**.

Some good backpropagation reference material, as well as references to further relevant background, may be found in the following sources: *The Handbook of Brain Theory and Neural Networks* (cited above); *Introduction to the Theory of Neural Computation*, by Hertz, Krogh, & Palmer (1991, Addison-Wesley, Redwood City, Calif.; and *Neural Networks for Pattern Recognition*, by C. M. Bishop (1995, Oxford University Press, Oxford, G.B.).

Numerous commercial software packages are available to assist in implementing backpropagation (and the rest of my preferred embodiment, in some cases) in computer software. One especially powerful and flexible one which is currently available for free (with certain copyright restrictions) is the PDP++ package by O'Reilly, Dawson, and McClelland. This package is available (at the time of this writing) from the Center for the Neural Basis of Cognition (a joint program between Carnegie Mellon University and The University of Pittsburgh) on the internet at http://www.cnbc.cmu.edu/PDP++/PDP++.html (also at http://einstein.lerc.nasa.gov/pdp++/pdp-user$_{13}$toc.html). The documentation for this package is also very useful for learning about backpropagation and its implementation using object oriented programming and the C++ language.

Regulation of Trials (Pattern Presentations)

The overall operation of the second preferred embodiment is illustrated in FIG. **11**. As with the first preferred embodiment, operation of this one can be viewed as a sequence of trials, each of which includes presentation of a single input signal **26**. Preferably the trials are divided into a set of training trials (i.e., with only training mode enabled), followed by a set of recognition trials (with only recognition enabled). One possible confusion here is that "recognition" is taken to include all operations performed with the backprop network—including training of the backprop net. Since this device does nothing special with respect to the backprop net, other than provide it with different inputs than it would otherwise have, backprop training will not be detailed, and is considered a "recognition" operation herein. I will specifically call it "backprop training" to distinguish it from "training", when necessary; the latter is meant to refer only to training of the unsupervised feature detectors **28** of my device.

Initialization Before Trials

Before any trials occur, the memory **40** is loaded with the training set. The memory **40** is implemented as a two-

dimensional array variable MEMORY[ ][ ], where the first dimension ranges over patterns, and the second ranges over elements within a pattern. Note that MEMORY[ ][n] corresponds to INPUT[n].

Preferably all patterns in the training set are stored in the memory **40**. However, if the training set is especially large, a non-preferred embodiment could be tried in which a random sample is chosen as the comparison patterns to store in the memory **40**. If so, patterns in the sample should be chosen independently and at random according to their distribution in the pattern domain.

The preferred features of the feature detectors **28** are implemented using the array WEIGHT[ ][ ]. The first dimension of WEIGHT ranges over the M feature detectors **28**, and the second ranges over the N input units. Note this is the reverse of the WEIGHT indexing scheme of the first preferred embodiment, because this embodiment is more naturally viewed as a pattern interpreter than a pattern generator (although both embodiments can be viewed in either way).

The weights must be initialized to small random values before any trials. Preferably they should be uniform random in the range 0.02 to 0.04, but this could be an experimental parameter if resources allow such experimentation. A possible improvement, which I have not tested, is to set the weight vector WEIGHT[*m*][ ] for each feature detector **28**[*m*] to a small multiple of (e.g. 0.01 times) a randomly selected training pattern, and then add a small random number (e.g. between 0.02 and 0.03) to each weight (such that positive weights always result). Note that no range limitation is placed on the weights during learning as in the first preferred embodiment, although the learning procedure itself will maintain the weights within the 0 to 1 range.

Training Trials

The operation over training trials is illustrated in more detail in FIG. **12**. On each training trial, a pattern is selected from the training set independently and at random according to the pattern domain distribution. Preferably this training pattern comes from the patterns stored in MEMORY. The training pattern is stored in the array TRAINPAT[ ].

Given a selected TRAINPAT, a loop is next performed over comparisons. Each comparison begins with the selection of a random pattern from MEMORY, and storage of it in the array COMPAREPAT[ ].

A test is performed on TRAINPAT and COMPAREPAT to determine if they are identical on every binary element. If so, COMPAREPAT is marked as "used" for this trial, and processing moves to the next comparison.

A second test determines whether TRAINPAT and COMPAREPAT differ only by "noise"; that is, whether they are "essentially identical". The definition of "noise" generally depends on the problem at hand, so to achieve optimal performance this test could be experimented with. If experimentation is not possible, however, my preferred test should be used, which is to reject differences with a Hamming distance (number of differing bits) of one. The purpose of this test, (which purpose should guide any experimental changes), is to reject those differences which don't represent a true feature of the pattern domain. If TRAINPAT and COMPAREPAT are judged to differ only by noise, COMPAREPAT is marked as "used" for this trial, and processing moves to the next comparison.

Another test is next performed which attempts to restrict the comparisons to differences of one, or at most a small number, of features. It is called the "dissimilarity test", because the goal is to throw out comparison patterns which are highly dissimilar from the training pattern. Ideally only pairs of patterns which differ by a single feature would be used, as these are best at indicating what the features of the pattern domain are. However, we can't identify the features a priori, so we can only use heuristics to guess at the number of differing features for a given pair of patterns.

My preferred dissimilarity test is to reject comparisons which have a Hamming distance greater than some fixed percentage of the number N of input units. I recommend using a value of 20%, as shown in FIG. **12**. However, if experimental resources permit, a crude optimization of this value should be performed. (Note that the Hamming distance used should never be less than or equal to that of the "essential identity" test, or else all comparisons would be rejected! Such excessive restriction must also be avoided if other, non-preferred tests are used.) This preferred value of 20% assumes that the input patterns are not sparse—that is, that on average a roughly equal number of pattern elements are on as are off. If this is not true, the preferred value should be computed by determining the average number of "on" bits in a pattern, over the entire training set, and using 40% of that average number.

It must be emphasized that this test will not be perfect, even with an optimized percentage. The problem is that a true feature could conceivably consist of a very large number of input units. However, the alternative—a method which considers every non-identical pattern pair to differ by a single feature—is much less theoretically justified. Also, as always, if the system designer has some reason to believe that a particular value would be more appropriate for a given pattern domain than my suggested value, the designer's informed guess is preferred as the starting point for experimentation.

Assuming COMPAREPAT passes the identity, near-identity, and dissimilarity tests, a difference vector is computed and stored as the variable DIFF[ ]. DIFF is obtained by the bit-wise operation AND-NOT. For two boolean variables x and y, the value of x AND-NOT y is true (equals one) if and only if x is true and y is false. Thus, each element DIFF[n] is set to the value of TRAINPAT[n] AND-NOT COMPAREPAT[n].

A loop is next entered over the M feature detectors **28**. For each such detector m, a variable DIST is computed which is the Euclidean distance between WEIGHT[*m*][ ] and DIFF[ ]. The minimum value of DIST over all feature detectors, and the index m corresponding to the minimum, are maintained in MIN and IMIN, respectively.

Once the minimum-distance feature detector **28**[IMIN] is found, its preferred feature WEIGHT[IMIN][ ] is moved toward the current difference vector DIFF[ ]. Note that DIFF represents a "part" of TRAINPAT for which feature detector **28**[IMIN] is taking responsibility.

The amount of learning done on each comparison is determined by LRATE, the learning rate. LRATE is equal to 1.0, times the reciprocal of the number of comparisons (including rejections) done on the trial (which equals NUMPATS, the number of training patterns, in my preferred embodiment), divided by ITRIAL, the index of the current trial (beginning with 1). For each element n of WEIGHT [IMIN][ ] and DIFF[ ], the difference DIFF[n]-WEIGHT [IMIN][n] is computed, and multiplied by LRATE, and the result added to WEIGHT[MIN][n].

The comparison loop continues in this fashion until all comparison patterns have been exhausted. New comparison patterns are chosen without replacement, so that each one from the comparison set in MEMORY is used once and only once for each TRAINPAT.

After all comparisons have been performed, and features updated, for this training pattern, a new training pattern is

selected. TRAINPATs, like COMPAREPATs, are not replaced in the pool once selected, so that each will be used once and only once, until all NUMPATS patterns have been used (at which point training may continue on a new cycle through the training patterns).

Stopping the Learning Process

A decision is made at some point to stop learning. My preferred method for this is to keep track, for each training pattern, of the number of comparisons on which each feature detector wins. That is, a 2-D array NUMWINS[ ][ ] is maintained, where NUMWINS[*m*][t] is the number of times feature detector m won a comparison on trial t. The entire training set is presented repeatedly (as indicated by the "recycle set as necessary" instruction in FIG. **12**), each iteration being as already described, until either (1) no element in the NUMWINS[ ][ ] array changes during the training set iteration, or (2) a maximum number of training set iterations is performed. The maximum could be experimented with, but my preferred value is 20.

Note that while this procedure requires multiple iterations through the training set, the trial index ITRIAL should not be reset, since it represents the total number of training trials which have occurred. Another index variable should be used to keep track of pattern presentations within a given training set iteration.

If experimental resources permit, it may be useful to try different criteria for stopping learning. This is especially true with large training sets, where learning may converge to an acceptable state within just one training set iteration. One such technique would be to maintain a running average for the MIN values (Euclidean distance between winning feature detector IMIN and the DIFF vector it wins), and stop learning when a plot of this running average reaches some criterion (small) slope.

Using the Backprop Network

Once training is finished, training mode is disabled and recognition mode is enabled. At this point, the particular backprop architecture and procedure employed will determine the order and manner in which patterns are selected. Recall that, as mentioned previously, training of the backprop network will now take place, but because the backprop network is a well-known module with respect to my device, all operations on it including training will be considered "recognition mode" herein.

All patterns in the training set must be converted for use by the backprop module, whether done all at once before training of the backprop net (as preferred, and as shown in FIG. **11**), or one at a time during its training. Once it is trained, new patterns to be recognized must also be converted to allow proper recognition.

The conversion of patterns may be viewed as an input layer feeding into the backprop net—albeit an input layer which (now) has fixed weights, and different activation functions than the backprop net. In the terminology of this specification, the feature activity signal **30** forms the input to the backprop module. Thus I describe here how to produce this signal **30** for this embodiment, and leave the implementation of backprop up to the user. The considerations that go into the particular implementation of backprop used are the same as those in prior art backprop nets, except as noted herein.

As shown in FIG. **11**, the feature activity signal **30** is stored as an array ACT[ ], and is determined as follows. The input pattern (signal **26**) is stored in the array INPUT[ ]. The value for a given ACT[j] is computed as the inner product between the INPUT[ ] and WEIGHT[j][ ] vectors. (The inner product is also known as the dot product, and is a measure of the similarity of the two vectors.)

Note that the ACT values will be real numbers, and may fall beyond the range 0–1; in particular, they may range as high as the number of elements in the input, N. Such real valued inputs are not a problem in general for backpropagation networks. However, there are some specialized implementations of backprop which assume or prefer binary inputs or inputs which are less than or equal to one. Such implementations of backprop would not be appropriate for this preferred embodiment.

The output of the backprop network, stored as the array OUTPUT[ ], becomes the output signal **36**. The activation values of the backprop network's output units might be used directly, e.g. as posterior probability estimates, or a classification index might be computed from them and used as the output signal **36** (in the latter case, OUTPUT[ ] would only be a one-element array). The exact method used depends on what type of effector **38** is used, and on the recognition problem being addressed; an appropriate method will be readily apparent to those skilled in the art, given a particular recognition task.

CONCLUSION, RAMIFICATIONS, AND SCOPE OF INVENTION

Thus the reader will see that a pattern recognition device according to the invention may be trained with fewer examples of physical patterns than prior art devices applied to the same task. Furthermore, the invention allows for improved generalization of learning given a relatively small training set. Still further, it allows for potentially improved scaling to relatively large architectures.

While my above description contains many specificities, these should not be construed as limitations on the scope of the invention, but rather as exemplifications of preferred embodiments thereof. Many other variations are possible. For example, neural network based embodiments could be used which are not strictly layered (i.e. have "layer skipping" connections), or which use some pattern of connectivity other than full connectivity between layers, such as limited receptive fields.

An embodiment similar to my first preferred embodiment might update weights simultaneously with Gibbs sampling; that is, each unit could be sampled and have its weights modified before moving on to another unit. More generally, a given feature detector **28**[*a*] may be modified by the updater **42** before assignment of another part to another feature detector **28**[*b*] takes place (this is true for virtually any other embodiment as well, including my second preferred embodiment).

Many other variations of the invention will become apparent to those skilled in the art, especially upon observing the relatively major differences between my two preferred embodiments.

Accordingly, the scope of the invention should be determined not by the embodiments illustrated and described, but by the appended claims and their legal equivalents.

What is claimed is:

1. A device for recognizing and responding to physical patterns, comprising:

(a) transducer means for producing an input signal representing a physical pattern in an environment;

(b) a plurality of feature detectors responsive to said input signal, each feature detector having weight means for storing a representation of a preferred feature, for producing a feature activity signal representing degrees to which each of said preferred features exists in said input signal, and for producing a feature description signal representing said preferred features;

(c) classifier means responsive to said feature activity signal, for producing an output signal representing a system action corresponding to said input signal;

(d) effector means responsive to said output signal, for committing an action in said environment;

(e) memory means responsive to said input signal, for approximately storing a representation of said input signal, and for producing a retrieval signal representing previously stored input signals;

(f) assigner means responsive to said input signal and to said retrieval signal and to said feature description signal, for producing a part mapping signal representing a mapping between a plurality of parts and at least one responsible feature detector, such that each part corresponds to a likely feature of said input signal and of said previously stored input signals;

(g) updater means responsive to said part mapping signal, for modifying each of said responsible feature detectors so as to make its preferred feature more similar to its assigned part;

whereby the modification of each of said responsible feature detectors is largely independent of the modifications of the other feature detectors;

whereby said device can be effectively trained with fewer physical pattern examples than a device having correlated feature training.

2. The device of claim **1** wherein said part mapping signal represents a mapping between said plurality of parts and a plurality of responsible feature detectors, and is such that each responsible feature detector has a high correspondence to its assigned part relative to the other feature detectors.

3. The device of claim **2** wherein said memory means is responsive to said feature activity signal and to said feature description signal, and said retrieval signal is dependent upon said feature activity signal and upon said feature description signal.

4. The device of claim **3** wherein said feature detectors, and said classifier means, and said memory means, and said assigner means, and said updater means comprise executable instruction code on a digital computing machine.

5. The device of claim **3** wherein said feature detectors are implemented with a neural network, such that the weight means for each feature detector comprises an array of modifiable connections configured for receiving said input signal.

6. The device of claim **5** wherein said neural network comprises executable instruction code on a digital computing machine.

7. The device of claim **5** wherein at least one unit of said neural network acts according to a noisy-OR function.

8. The device of claim **7**, further including means for storing a contribution to the activation probability of said at least one unit, such that said contribution may be accessed on a plurality of activation cycles.

9. The device of claim **8** wherein said contribution is a sum over each inactive child unit of a negative logarithm of a quantity representing one minus the weight from said at least one unit to said inactive child unit.

10. The device of claim **5** wherein said assigner means is configured to perform a soft segmentation of said input signal to obtain said parts.

11. The device of claim **2** wherein said memory means is a lossless storage device.

12. The device of claim **11** wherein each of said parts is a difference vector representing a difference between said input signal and a previously stored comparison pattern represented by said retrieval signal.

13. The device of claim **12** wherein said assigner means is configured to assign each of said parts to a winning feature detector, said winning feature detector having the preferred feature which has a minimum distance from said difference vector.

14. The device of claim **2** wherein said updater means is configured to modify each of said responsible feature detectors so as to make its preferred feature move to a new input space location which is substantially along the vector from its current input space location to the input space location of its assigned part.

15. A method for creating a pattern recognition device, comprising the steps of:

(a) providing transducer means for producing an input signal representing a physical pattern in an environment;

(b) providing a plurality of feature detectors responsive to said input signal, each feature detector having weight means for storing a representation of a preferred feature, for producing a feature activity signal representing degrees to which each of said preferred features exists in said input signal, and for producing a feature description signal representing said preferred features;

(c) providing classifier means responsive to said feature activity signal, for producing an output signal representing a system action corresponding to said input signal;

(d) providing effector means responsive to said output signal, for committing an action in said environment;

(e) providing memory means for approximately storing input patterns, and for producing a retrieval signal representing previously stored input patterns;

(f) using said memory means to approximately store a sequence of comparison patterns;

(g) providing a training pattern;

(h) identifying a plurality of parts in said training pattern, such that each part corresponds to a likely feature of said training pattern and of said comparison patterns;

(i) assigning each of said parts to a corresponding responsible feature detector;

(j) modifying each of said responsible feature detectors so as to make its preferred feature substantially directly more similar to its assigned part;

(k) training said feature detectors by repeating steps (g) through (j) on a significant portion of a training set until a training criterion is reached;

whereby the modification of each of said responsible feature detectors is largely independent of the modifications of the other feature detectors;

whereby said method allows effective creation of a pattern recognition device with fewer pattern presentations than a device having correlated feature training.

16. The method of claim **15**, further including the steps of:

(l) repeating steps (a) through (d) to create a comparable pattern recognition device;

(m) transferring the preferred feature of at least one of the trained feature detectors to at least one corresponding feature detector of said comparable pattern recognition device.

17. The method of claim **15** wherein said memory means is responsive to said feature activity signal and to said feature description signal, and said retrieval signal is dependent upon said feature activity signal and upon said feature description signal.

18. The method of claim **17** wherein said feature detectors are implemented with a neural network, such that the weight

means for each feature detector comprises an array of modifiable connections configured for receiving said input signal.

19. The method of claim **18** wherein at least one unit of said neural network acts according to a noisy-OR function.

20. A device for recognizing and responding to physical patterns, comprising:

(a) a transducer capable of producing an input signal representing a physical pattern in an environment;

(b) a plurality of feature detectors each responsive to said input signal, each feature detector having weight storage capable of representing a preferred feature, each of said feature detectors being capable of producing a feature activity signal element representing a degree to which its preferred feature exists in said input signal, and being capable of producing a feature description signal element representing its preferred feature;

(c) a classifier responsive to each said feature activity signal element, capable of producing an output signal representing a system action corresponding to said input signal;

(d) an effector responsive to said output signal, capable of committing an action in said environment;

(e) a memory responsive to said input signal, capable of approximately storing a representation of said input signal, and capable of producing a retrieval signal representing previously stored input signals;

(f) an assigner responsive to said input signal and to said retrieval signal and to each of said feature description signal elements, capable of producing a part mapping signal representing a mapping between a plurality of parts and a plurality of responsible feature detectors, such that each part corresponds to a likely feature of said input signal and of said previously stored input signals, and such that each responsible feature detector has a high correspondence to its assigned part relative to the other feature detectors;

(g) an updater responsive to said part mapping signal, capable of modifying each of said responsible feature detectors so as to make its preferred feature vector move substantially directly toward its assigned part vector;

whereby the modification of each of said responsible feature detectors is largely independent of the modifications of the other feature detectors;

whereby said device can be effectively trained with fewer physical pattern examples than a device having correlated feature training.

* * * * *